Reification of Network Resource Control in Multi-agent Systems

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Chen Liu

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

In *multi-agent systems* [1], coordinated resource sharing is indispensable for a set of autonomous agents, which are running in the same execution space, to accomplish their computational objectives. This research presents a new approach to network resource control in multi-agent systems, based on the *CyberOrgs* [2] model. This approach aims to offer a mechanism to reify network resource control in multi-agent systems and to realize this mechanism in a prototype system.

In order to achieve these objectives, a uniform abstraction *vLink* (Virtual Link) is introduced to represent network resource, and based on this abstraction, a coherent mechanism of vLink creation, allocation and consumption is developed. This mechanism is enforced in the network by applying a fine-grained flow-based scheduling scheme. In addition, concerns of computations are separated from those of resources required to complete them, which simplifies engineering of network resource control. Thus, application programmers are enabled to focus on their application development and separately declaring resource request and defining resource control policies for their applications in a simplified way. Furthermore, network resource is bounded to computations and controlled in a hierarchy to coordinate network resource usage. A computation and its sub-computations are not allowed to consume resources beyond their resource boundary. However, resources can be traded between different boundaries.

In this thesis, the design and implementation of a prototype system is described as well. The prototype system is a middleware system architecture, which can be used to build systems supporting network resource control. This architecture has a layered structure and aims to achieve three goals: (1) providing an interface for programmers to express resource requests for applications and define their resource control policies; (2) specializing the CyberOrgs model to control network resource; and (3) providing carefully designed mechanisms for routing, link sharing and packet scheduling to enforce required resource allocation in the network.

Acknowledgements

This thesis is the result of two years of work whereby I have been accompanied and supported by many people. It is a pleasure that I have an opportunity to thank all of them.

I would like to express my sincere appreciation and gratefulness to my supervisor, Professor Nadeem Jamali. Without his invaluable guidance, suggestion and encouragement, this thesis would be impossible. His enthusiasm and integral view on research has made a deep impression on me, and has important influence on my academic pursuit. I feel very lucky to have studied under his supervision and what I learned from him will benefit my whole academic career. In addition, Professor Jamali also gave valuable suggestions and helped in my personal life.

I would like to thank Professor Derek Eager and Professor Michael Horsch, the two professors in my thesis committee. I very much appreciate it that Professor Eager spent time meeting with me and giving valuable suggestions on my work. With his suggestions, my understanding of networking research has broadened. I also benefited a lot from his Networking and Performance Analysis courses. In addition, I am very grateful for Professor Eager's thorough and detail comments on both of my proposal and thesis drafts. Professor Horsch was willing to be the committee member when his schedule was very tight. I am also thankful for his valuable suggestions on my thesis, as well as his encouragement which helped my confidence during my thesis defence.

Here, I would like to express many thanks to Professor Aryan Saadat Mehr's willingness to be the external member of the committee, and his helpful suggestions on my work.

I also would like to give thanks to Professor Kevin Schneider. He met with me when he was very busy, and his insightful suggestions helped me to improve my system design. In addition, his Software Engineering class benefited me a significantly in software architecture design. Many thanks go to Niklas Carlsson for the insightful discussion with me. His suggestions helped me avoid some dead ends.

The Department of Computer Science at University of Saskatchewan offered a friendly environment and sufficient resources for me as a graduate student. I am very thankful to all office staff and their hard work. Special thanks go to Jan who is Mom for all graduates in the department and gave us mother-like love. Her hard work makes our life much easier.

My friend and colleague Xinghui Zhao offered valuable support to this work. My discussions with her on CyberOrgs helped me understand this model from another point of view. During the writing of this thesis, she reviewed a number of chapters and gave useful suggestions on both grammar and organization. I appreciate her helpful comments. I also would like to thank Hongxing Geng for lending me use his computer to do experiments. I also would like to express my thank to Weidong Han and Tianping Ter for spending time disscussing technical problems with me.

I would like to express my gratitude to my beloved parents. It is the example of my father's

wisdom and strong will power that supported me pass through my difficult times. Finally, I want to express my gratitude to Chen Dai, my good friend, who was with me and supported me during my hardest time.

Contents

List of Tables

List of Figures

List of Abbreviations

AA	Actor Architecture
AF	Assured Forwarding
API	Application Programming Interface
BFS	Breadth First Search
CLVL	Controlled Loss Virtual Link
DFS	Depth First Search
DiffServ	Differentiated Service
EF	Expedited Forwarding
FCFS	First Come First Serve
GPS	Generalized Processor Sharing
IntServ	Integrated Service
ISPN	Integrated Services Packet Network
JVM	Java Virtual Machine
JDK	Java Development Kit
RM API	Java Resource Management Application Programming Interface
LOF	List of Figures
LOT	List of Tables
MAS	Multiagent Sysmtems
PHB	Per-hop Behavior
PLAN	Packet Language for Active Networks
QoS	Quality of Service
RSL	Resource Specification Language
RSVP	Resource ReSerVation Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WFQ	Weighted Fair Queueing

Chapter 1

Introduction

In the past few years, *multi-agent systems* have offered an appealing paradigm for creating systems that operate in open and distributed environments, and have been widely applied to a number of applications. In order to design systems in which agents interact with each other productively, and collaborate for problem solving, the issue of resource management arises, which is the area of this thesis.

1.1 Background and Problem

This section first introduces background work and the motivation for this research. Then, a summary is presented of the proposed approach and contributions.

1.1.1 Multi-agent Systems

In the literature, various definitions have been proposed for the term *multi-agent system*. Generally speaking, a multi-agent system is characterized as a system composed of autonomous agents where (1) each agent has incomplete information or capabilities for solving the problem; (2) there is no system global control; (3) data is decentralized; and (4) computation is asynchronous [1]. The multi-agent system paradigm offers a natural way to model *Open Systems* [3] which consist of many distributed, asynchronous components that are open to interact with the environment. It also provides an infrastructure for developing interaction protocols and coordination models by which agents can interact with each other productively and solve problems coordinatively. In the context of multi-agent systems, a complex problem is often decomposed into sub tasks which are then carried out by distributed agents asynchronously. Coordination between agents, which are running in a shared resource environment, is required to achieve a better result [4]. This requirement highlights the importance of having resource management mechanisms in multi-agent systems.

1.1.2 Resource Problem in Multi-agent Systems

Resource management is an important issue for multi-agent systems. By *resource*, it is meant computational resources which are necessary for carrying out an action, such as processor time, memory, and network bandwidth etc. In a multi-agent system, agents are fueled by resources to perform their actions. The multi-agent systems of interest to this research have bounded resources, which are shared by a number of self-interested or cooperative agents. Since available resources are limited, agents may compete with one another for accessing these resources and result in conflicts [5]. Specifically, it is possible that a small number of agents monopolize most of the resources, and other agents which are sharing the same resource space may not have opportunities to progress. For example, a malicious agent may spawn a large number of agents to consume the system resources in an exponential way [2]. In addition, even for currently executing agents, some may seize more resources, causing performance degradation for other agents. Such unrestricted competition for resources can be detrimental to the execution of individual agents and the system as a whole. Therefore resource consumption in multi-agent systems needs to be controlled in a coordinated way.

This thesis focuses on network resource control in multi-agent systems. Due to the absence of global knowledge, each agent needs to interact with one another to exchange information. The basic way of interaction in a multi-agent system is communication, in the form of asynchronous message sending. Network resources are consumed during the transfer of these messages, when agents are located at distributed nodes connected by an overlay network. Here, *network resources* is defined as the access right to an overlay link and an amount of bandwidth of the link. In the rest part of this thesis, terms 'link' and 'network' herein are used interchangeable with 'overlay link' and 'overlay network' respectively. In a bounded resource environment of multi-agent systems, network bandwidth is the contended resource to agent communications. Without proper coordination in bandwidth consumption, not only is it possible that communication tasks cannot be done, which influences the achievement of computational objectives for applications, it is also possible that network overload and traffic congestion may happen in the network.

In an overlay network, a link has limited capacity (maximum link bandwidth) which restricts the amount of data transferred in a time unit. If applications are not aware of this restriction, they may send data at a faster rate causing overload. In addition, despite a link having sufficient capacity, it may be possible that one or more flows monopolize the link, hindering other communications. It is also possible that flows interfere with one another when passing through the same link. We define a *flow* as a stream of data belonging to the same communication which is being transferred through a network. These problems arise because, in a network, resources are not allocated to meet individual application's needs, but to maximize the utilization of overall network. The discrepancy between resource control requirements at the application level and resource allocation policies at the network

level is a major challenge to network resource control in multi-agent systems. This research aims to reconcile this discrepancy and coordinate network resource consumption in multi-agent systems.

1.2 Research Objective and Approach

The objective of this research is to offer an effective mechanism to coordinate network resource usage in multi-agent systems and provide an effective system to enforce this mechanism. In this work, a middleware architecture is developed, which provides an infrastructure and facilities to control network resources in multi-agent systems. This architecture enables programmers to build systems which allow applications to express resource requests, and gives programmers the privilege of controlling resource allocation. It is assumed that all resources are allocated by the proposed system. Consequently, it is not necessary to worry about resource competition from application outside this system. In this section, the proposed approach is described from two levels: the application level and the network resource control level.

At the application level, the first objective guiding the proposed approach is to simplify the programmers' tasks in resource control. This objective is achieved by separating resource concerns from functional concerns of computations. In the model of CyberOrgs, computations are carried out by actors, which are primitive agents. A cyberorg is the basic unit of resource control, which is an abstraction defined separately from actors. With this separation of concerns, programmers are allowed to code resource control separately from application functions. Furthermore, in order to provide a simplified and uniform representation of network resource, a *Virtual Link* abstraction is introduced, using which attributes of network resources are characterized. Aided by this abstraction, programmers are enabled to describe their resource requirements in terms of attributes, rather than concrete configurations. The second objective is to coordinate resource usage among computations. A cyberorg bounds resources that can be used by an actor. Each cyberorg has the right to allocate resources to its local actors, and to give some of its resources to others. A market mechanism is introduced to enable resource trading and transferring between different cyberorgs. The model of CyberOrgs is described in detail in Chapter 3.

At the network resource control level, the objective of the proposed system is to employ appropriate resource discovery and allocation mechanisms to realize resource control requirements at the application level. In comparison to this obligation, current networks only provide best-effort services, in which individual application's requirements are not considered. For example, the TCP congestion control mechanism adjusts traffics according to the current congestion level in the network, which aims to fully utilize the network, rather than to satisfy an individual flow's requirements. This proposed approach is to carefully design schemes of routing, flow multiplexing and packet scheduling, to find a 'qualified' path to meet an application's bandwidth requirement

and allow different flows to share a single link, and to control the sending rate of packets from each flow in order to enforce bandwidth allocation.

1.3 Contributions

This work makes contributions in the area of network resource management, as follows.

- The CyberOrgs model is first specialized in network resource control.
- An abstraction of network resource is introduced to provide a simplified and uniform representation of network resource, which serves as a foundation for the development of a resource allocation mechanism.
- A fine-grained flow-based scheduling scheme is developed to enforce required resource allocation in the network.
- An effective system to coordinate network resource usage in multi-agent systems is developed.

1.4 Outline

The rest of this thesis is organized as follows:

Existing work in related areas is reviewed in Chapter 2. Chapter 3 describes the proposed reification of network resource control. System design and implementation issues are presented in Chapter 4 and 5, respectively. In Chapter 6, experiment results are discussed. Finally, conclusions and future work are identified in Chapter 7.

Chapter 2

Related Work

In this chapter, previous work relating to this research is discussed. First the issue of resource management in three disciplines is discussed: multi-agent systems, distributed systems and networking. Section 2.1.1 describes resource management in multi-agent systems, and three types of approaches are summarized. In Section 2.1.2, related projects carried out in grid computing are presented, and previous research of network resource management in networking is studied in Section 2.1.3. Finally, a number of packet scheduling algorithms are presented in Section 2.2.

2.1 Resource Management

In this section, related work in three disciplines is summarized, including multi-agent systems, distributed systems and networking.

2.1.1 Resource Management in Multi-agent Systems

Resource management issues arise in multi-agent systems due to unrestricted competition over finite resources among agents. Especially, in the situation where agents may migrate to other machines seeking a better execution environment, problems with guaranteeing security and quality of service can arise because of this flexibility. In this section, a number of Java-based approaches are reviewed, including JRes [6], JSeal2 [7] and Java RM API [8]. Due to performance issues in extending Java, some projects try to modify the Java virtual machine or develop new virtual machines to build in resource management mechanisms. Nomads [9] is an example of this trend. Formal models for bounded resource computation include Quantum [10, 11] and CyberOrgs [2].

JRes

JRes [6] is a resource management interface for Java, which allows the accounting and limiting of resources such as processor time, heap memory, and network bandwidth. Aiming to support the creation of portable and extensible web environments, JRes allows untrusted code to query information including resource limit and usage, and enables authenticated trusted code to manage and enforce resource limits.

In JRes, resource accounting requires native code support and bytecode rewriting. For CPU usage, an operating system level handle is created for each thread. A special polling thread periodically uses these handles to query the system for CPU usage. Network resource accounting is achieved by rewriting the java.net package, which contains native classes that can access the network directly. Promised by this rewriting approach, corresponding information is recorded whenever a native network-using method is invoked. Memory accounting also relies on bytecode rewriting. The code of every method is changed by inserting appropriate bytecode in the original method code, whenever an object allocating instruction occurs.

JRes is simple and flexible to compose complex resource management policies. However, the major disadvantage of JRes is the per-thread resource control. This approach complicates resource accounting, and can not handle resource sharing by threads either. In addition, JRes only provides resource control to traditional computational resources.

JSeal2

JSeal2 [7] is a secure mobile agent system, in which resource control is supported to implement preventive mechanisms against denial of service caused by hostile or poorly implemented mobile code. Similar to JRes, JSeal2 also provides APIs for accounting and limiting the usage of resources. However, besides physical resources, JSeal2 controls logical resources such as the number of threads and number of protection domains.

In JSeal2, the basic unit of resource control is a Seal, instead of an individual thread. A Seal may be a mobile object or a service and executes in its own protection domain, sharing no state with other seals. In addition, seals in JSeal2 are organized in a hierarchy, similar to cyberorgs in the CyberOrgs model. At the system setup phase, RootSeal, the first domain possesses all resources the Java runtime system allocates from the underlying operating system. When a child seal is spawned, the creator seal donates part of its resources to the new seal. Furthermore, JSeal2 also allows a child seal to share resources owned by its creator seal.

For the purpose of complete portability, JSeal2 employs the bytecode rewriting technique for the accounting of both CPU time and memory resources. Instead of modifying the Java run time system, bytecode is modified before being loaded by the JVM. CPU time accounting is achieved by counting the number of executed bytecode instructions. Therefore code for CPU accounting is inserted in every basic block of code. Similarly, code for memory accounting is inserted before each memory allocation instruction.

Java RM API

Java RM API [8] was proposed as an extensible, flexible and widely applicable resource management interface for the Java ^{TM} platform. The interface is capable of modeling a variety of resources and resource management policies. In Java RM API, resources are defined by describing a set

of properties. With this abstraction, users are able to define application-specific resources, and traditional computational resources such as CPU time and heap memory as well. The unit of resource control in Java RM API is an isolate, an encapsulated program or application component. There is no shared state between isolates and isolates are organized in a hierarchical way. Each isolate is bound to a resource domain which represents a resource consumption policy. A dispenser is in charge of monitoring available resources to computations and serves as a bridge between the resource management interface and the resource implementation.

The resource management framework of Java RM API is implemented on top of the Isolate API. Although this prototype does not modify the underlying virtual machine, the task of exposing resources through RM API requires the modification to the Java Virtual Machine and Java Development Kit. Compared with JSeal2 and JRes, Java RM API allows programmers to define their own resources. In addition, as an extension to Java, code written in the RM API is portable.

Nomads

Nomads [9] is a mobile agent system which provides the ability to control agent usage of resources. This ability helps to achieve the goal of safe Java agent execution. Nomads has a new Java compatible Virtual Machine (VM), which is called *Aroma VM*, and mechanisms for monitoring and controlling resource usage are built in Aroma. Aroma is implemented in C++ and has two parts: a VM library and a native code library. The VM library can be linked to other application programs, and the native code library implements the native methods in Java API and is dynamically loaded by the VM library.

The implementation of enforcing resource limits is part of the native code library's responsibility. The current version of the library only enforces disk and network limits in terms of rate, quantity and space. Rate limits guarantee that the input and output rate of a program does not exceed the specified value. The average rate is measured by dividing the number of bytes written and read to the network and disk by the elapsed time. Quantity limits control the total bytes read or written to the network and disk, and space limits only controls the disk space usage.

In Nomads, each agent is running on an individual Aroma VM. Not only does this allow strong mobility of agents, but resource accounting and control is also simplified. However, having a special virtual machine also introduces trade-off between new features (such as strong mobility and resource control) and good performance. In addition, compatibility issues may also prevent the employment of new virtual machines.

Quantum

Quantum [10] is a theoretical model developed for controlling resource consumption in distributed computing. This model was first proposed in 1997 and was extended in [11] to support controlling distributed and multi-type resources. There are three key ideas in Quantum, which are group

creation, asynchronous notification and energy transfer.

In Quantum, resources are represented by *energy*, and are consumed by computations. A *group* is a unit of resource control, in which quotas of energy can be associated with computations. Groups in Quantum are organized in a hierarchical way, therefore a group is able to create a subgroup and sponsor the new group by donating part of its energy to the new group. In order to inform the termination of a computation and exhaustion of energy, asynchronous notification is used. When the computation sponsored by a group is finished, and this group has no subgroups, a termination function is asynchronously called. Remaining resources of the terminated group are released to its parent group. Similarly, when a group does not have enough energy to sponsor the computation associated with it, an exhaustion function is invoked and the rest of resource belonging to this calling group is given back to its parent group. Resource transfer between groups is supported by two primitives: *pause* and *awaken*. Pause causes a running group and its subgroups to be exhausted, and transfer resources belonging to this hierarchy to the group which invoked and sponsored this pause action. On the contrary, awaken supplies a group with an amount of energy and changes the state of an exhausted group to running. Compared with the proposal approach of this research, Quantum does not supports distributed resource control unit and market mechanism for resource trading.

2.1.2 Resource Management in Grid Computing

In distributed systems, system heterogeneity is the major hurdle for resource sharing. How to harness the power of idle computational resources belonging to different systems is a question of interest. This section focuses on resource management for grid computing, which emerged in the mid-1990s. Underlying the Grid concept, the specific problem is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [12]. Virtual organizations are dynamic collections of individuals, institutions, and resources. Grid technologies are seeking integrated approaches for shared resource usage among heterogeneous distributed systems. This section presents two major projects in this field, Condor [13] and Globus [14, 15].

Condor

Condor [13], developed in 1987, was one of the earliest systems which enabled pooling of workstations in a network for solving large problems. Considering the situation where the utilization of workstations is unbalanced in different systems, Condor aims to take advantage of under utilized workstations with minimal influence on the activities of people who own workstations. In brief, Condor schedules remote jobs on idle workstations in the background, and checkpoints remote jobs on the workstation whose owner becomes active, and transfers these jobs to another workstation.

In a Condor system, each workstation has a scheduling index which determines its priority to

access remote cycles. The index of a workstation is increased when remote capacity is allocated to the station, and is decreased when the station's request of accessing remote cycles is refused. In the system, a central coordinator periodically checks whether any station has jobs to schedule and assigns capacity to the high priority station which has a job. In case that there are no idle workstations in the system, the coordinator preempts a remotely executing job from a station with a lower priority and assigns the newly available capacity to the high priority station. Each workstation knows the relative priority of the jobs and schedules them accordingly.

Although Condor is successful in maximizing the utilization of workstations, this approach is limited in two ways. Firstly, it is only designed for CPU time sharing. Secondly, unlike approaches applied in multi-agent systems, Condor is a specific resource allocation policy rather than a generalized mechanism.

Globus

The Globus [14] project supports the construction of *metacomputers* for high performance applications. Metacomputers are execution environments in which high-speed networks are used to connect supercomputers, databases, scientific instruments, and advanced display devices, perhaps located at geographically distributed sites. Due to the issues of resource diversity and heterogeneity, Globus is developing a toolkit for sharing and accessing large and possibly heterogeneous resources over a network. The infrastructure of Globus provides a resource management architecture, which can be used to construct a range of global resource management strategies [15].

In Globus resource management architecture, applications can express resource requests using an *extensible resource management language* (RSL). These high-level RSL expressions are transformed by *resource brokers*, which implement domain-specific resource discovery and selection policies, into a set of separate and more specific resource allocation requests. Then, a *resource co-allocator* dispatches each request to the appropriate local resource manager, *GRAM*. Each GRAM is responsible for a particular local set of resources. GRAM serves as a connection between high-level resource allocation requests and individual resource management systems.

2.1.3 Network Resource Management

In networking, how to allocate resources in networks is an important issue, and much study has been carried out in this subject. In this section, existing technologies in two aspects of resource allocation are discussed [16]: congestion control and Quality of Service.

Congestion Avoidance and Control

Traffic congestion is a major issue which may result in longer delay, packet loss and inefficient usage of resource in networks. This issue is closely related to the availability of network resources. If

currently available resource in the network is inadequate to accommodate incoming traffic, the network will be overloaded. Congestion control describes the efforts made by network nodes to prevent or respond to overload conditions. This section focuses on TCP congestion control mechanisms.

There are four algorithms applied in TCP for congestion control including the *slow start*, *congestion avoidance*, *fast retransmit* and *fast recovery* [17, 18]. Slow start is used in two scenarios. First, this algorithm is applied at the beginning of a connection, when the sender does not know how much data is going to have in transit in a given time. The second scenario occurs when timeout occurs. In a slow start procedure, TCP slowly probes the network with unknown conditions to determine the available capacity. This algorithm is used to avoid congesting the network with a large burst of data. As slow starts repeatedly increases the load of data it sends to the network until the point at which congestion occurs, congestion avoidance is activated. During the phase of congestion avoidance, network is asserted to be congested if packet loss is detected and the sender's window size is decreased multiplicatively, (which becomes an exponential decrease over time if the congestion persists) [17]. This mechanism adjusts network utilization based on the current congestion level. Fast retransmit and fast recovery are used to speed up the sender recovery from packet loss and adapt to equilibrium. In specific, the TCP sender retransmits the segment which appears to be missing after the arrival of 3 duplicate ACKs, without waiting for the retransmission timer to expire. After the fast retransmit algorithm sends the missing segment, the fast recovery algorithm controls the transmission of new data using linear increase of the congestion window until a non-duplicate ACK arrives [18].

Although TCP congestion control is used to adjust the behavior of the sender according to the current congestion condition, the major purpose is to maximize the overall network utilization. Individual flow requests are not taken into consideration.

Network Quality of Service

Network Quality of Service provides service assurances to satisfy different requirements from applications, such as loss, delay and throughput, which are more than what congestion control promises. The current Internet architecture only provides point-to-point best effort services, which can not satisfy applications such as remote video, multimedia conference and so on. More service guarantees and qualities of service are demanded. Research in this area is summarized here into two levels: the network level and application level. Approaches applied at the network level attempt to establish new architectures for the Internet and require changes to the IP layer which is difficult to realize at the current stage. Examples are IntServ [19], DiffServ [20] architectures and Q-RAM-based QoS model [21]. By contrast, approaches at the application level, such as Overlay-based QoS and Active Networks, avoids changing IP layer by adding advanced mechanism to upper layers. Details of these projects are discussed in this section. Compared with these work, this research employes

a fine-grained, flow-based network resource allocation without changes to the IP layer.

Integrated Service (IntServ) The IntServ architecture is proposed to support real-time applications, which require a bound (either statistical or absolute) on the delivery delay of each packet in an Integrated Services Packet Network (ISPN) [19, 22]. According to different tolerance degrees of applications, the IntServ architecture provides two service commitments: guaranteed and predictive services. Guaranteed service is proposed to provide strict upper bound on delay to intolerant application, while predictive service may allow some violation of the delay bound for tolerant application.

The architecture of IntServ is composed of four components which are:

- Flow specification, allows users to characterize the traffic and give QoS requirements. Flow spec allows Internet hosts to negotiate with the Internet for rights to use a certain part of the Internet's resources. In [23], a proposed flow specification data structure contains traffic characteristics, sensitivity to delay, sensitivity to loss and desired service guarantee type. Another flow spec is proposed by ST-II (Internet Streaming protocol, version 2) [24], to describe the required characteristics of a stream, including bandwidth, delay, and reliability parameters.

- Admission control, determines which resource request to grant and which to deny based on a number of criteria, such as resource availability, higher network utility and influence on prior service commitments. For guaranteed services, reservation-based admission control algorithms are used to allocate resources to a flow depending on a priori traffic characterization. This approach may maximize network utilization if traffic is precisely characterized. Recently, a measurement based approach [25] is proposed to predictive services, which aims to achieve higher network utilization. In this approach, a priori characterization only applies on incoming flows, and measurements are used to characterize flows, which have already been accepted and in place for a certain period.

- Resource reservation, sets aside certain amount of resources in order to guarantee the required quality of service for a particular flow. A reservation setup protocol, RSVP [26, 27], is developed to work on behalf of applications to request through the network and make resource reservation at nodes. RSVP is receiver-oriented, i.e. the receiver is responsible for resource reservation.

- Packet scheduling, is the most important component in the network architecture because it determines which qualities of service the network can provide. A number of packet scheduling algorithms are reviewed in Section 2.3.

The IntServ architecture supports per-flow QoS guarantee, which demands all flows in the network have to be restricted, otherwise it is not possible to make QoS guarantees for a particular flow. This demand prevents IntServ to be employed in the Internet, which is an open environment composed of numerous autonomous systems. Neither is it reasonable, nor it is realistic to require all systems employ the IntServ architecture. In addition, IntServ imposes too much complex work on network routers which may slow them down.

Differentiated Service (DiffServ) DiffServ is another architecture proposed to improve QoS guarantees in the Internet. Similar to IntServ, DiffServ also aims to guarantee QoS requirements for applications. However, rather than treating individual flows differently, DiffServ guarantees quality of service to a traffic aggregate, which is a bundle of flows. In addition, service provisioning and traffic conditioning are pushed to the edge of the network in order to achieve scalability [20]. This architecture is composed of three main elements: Per-hop behaviors (PHB), packet classification and traffic conditioning functions.

A per-hop behavior (PHB) is a description of the externally observable forwarding behavior (i.e., loss, delay, jitter) of a DiffServ node applied to a particular DiffServ behavior aggregate [28]. The PHB defines how a traffic aggregate is treated at individual network nodes. Two standardized PHBs are Expedited Forwarding (EF) [29] and Assured Forwarding (AF) [30]. EF PHB provides a building block for low loss, low delay, and low jitter services. AF PHB group is a means for a provider DS domain to offer different levels of forwarding assurances for IP packets received from a customer DiffServ domain.

A packet classification policy uses DS codepoint remarking within the DS domain to identify the subset of traffic which may receive a differentiated service. DS codepoint is a specific value used to select a PHB.

Traffic conditioning performs metering, marking, shaping, and policing to ensure that the traffic entering the DS domain conforms to the rules specified in TCA (Traffic Conditioning Agreement) [31].

Q-RAM-based QoS Model Q-RAM-based QoS model [21] aims to establish a new network architecture which enforces appropriate resource allocation in order to guarantee quality of service. In their work, network bandwidth is considered separately from delay and packet loss and is achieved by optimized route discovery and bandwidth reservation. This model applies the hierarchical network structure and uses negotiation between different networks to find out an optimized route.

Overlay-based Quality of Service Difficulties in changing the IP infrastructure hinders the adoption of IntServ and DiffServ in today's network. Another trend of Network QoS is overlays

based QoS. This approach claims that QoS can be achieved without support from the IP routers. OverQoS [32], is an example, which uses an abstraction CLVL (controlled loss virtual link) to bound the loss rate of a traffic aggregate and provide services of smoothing packet losses, prioritizing packets within an aggregate and statistical loss and bandwidth guarantees.

Active Networks Active networks research is a novel approach to network architecture in which the switches of the network perform customized computations on the messages flowing through them [33]. This approach allows users to specify their application-oriented control requirements and build up user-aware networks, including resource allocation. PLAN [34] (Packet Language for Active Networks) is a new language for programs that form the packets of a programmable network. PLAN is based on the simple typed lambda calculus and provides a restricted set of primitives and data types. In PLAN, resource boundary is taken into account. They use hop number to specify the resource bounded to each flow, and resource bound to restrict the maximum resource can be consumed by each packet on a node.

2.2 Packet Scheduling

In the proposed implementation, bandwidth allocation is achieved by packet scheduling. An appropriate packet scheduling algorithm is needed to determine which packet to be sent next and when to send it. Existing technologies in this area are reviewed in this section.

2.2.1 First Come First Serve

The First Come First Serve (FCFS) algorithm is the simplest scheduling algorithm, which buffers packets from different sessions in a packet queue according to their arriving order. A packet p cannot be scheduled until all backlogged packets before it are sent out. This algorithm is reasonable if all previous packets are in the same session as p is. However, if the previous packets are from different sessions, packet p may be punished because packets belonging to some other sessions may arrive earlier as a burst.

2.2.2 Strict Priority

In a priority scheduling, packets are treated differently according to their priorities. A lower priority class is scheduled only after higher priority classes have no packets waiting for service. Obviously, higher priority classes get lower delay, better throughput and lower loss. However, it might end up that lower priority classes are starved when higher classes are overloaded.

2.2.3 Round Robin

Round Robin attempts to treat all sessions equally. Arriving packets are queued up according to sessions. In each scheduling cycle, the same number of packets in each queue is sent out in a fixed order. The cyclical discipline in round robin scheduling ensure that no single session can dominate the attention of the server at the expense of other sessions. However, this scheduling algorithm can not treat certain sessions better than others.

2.2.4 Weighted Round Robin

Considering the inflexibility of Round Robin scheme, a Weighted Round Robin attempts to treat each session differently. For this purpose, a weight is associated with a session. Let the weight of session i be w_i. When the server polls a session i, w_i fixed size packets will be sent out before moving to the next session in the scheduling cycle. However, there is a tradeoff between flexibility and packet delay, since a large range of weights leads to large service cycle times.

2.2.5 Generalized Processor Sharing

Although Weighted Round Robin gives different treatments to different sessions, it is difficult to bound packet delay. Generalized Processor Sharing (GPS) [35, 36] is reputed to be an efficient, flexible and analyzable scheme which guarantees delay and throughput. Generalized Processor Sharing (GPS) allows flows to have different service shares in accordance with their desired quality of service, and guarantees the minimum level of service which an individual flow receives independent of the behavior of by other flows. When a packet from session i arrives, the virtual time is updated and the packet is stamped with the virtual finishing time. The server is work conserving and serves packets in an increasing order of time stamp. Weighted Fair Queueing (WFQ), an approximation of GPS, aims to allow different flows to share the same link and have different guaranteed bandwidth allocated to them. Compared with WRR, WFQ allocates relative percentage of bandwidth to an individual flow. Let B_i be the bandwidth allocated to flow i, W_i be the weight assigned to flow i, and C is the capacity of the shared link. The bandwidth guarantee formula is in the form of

$$B_i = \frac{W_i}{\sum_j W_j} C.$$

This mechanism ensures that the bandwidth guarantee for each flow is independent and is not influenced by other flows. Besides, this scheme also allows configurable number of flows and guarantees delay and throughput as well.

Chapter 3

Network Resource Control Reification

A general description of the proposed approach for this research is stated in Section 1.2. This chapter introduces the theoretical foundation of the proposed approach to network resource control. First, two formal models on which this work is based are described: Actors [37] and CyberOrgs [2]. The proposed approach to reification of network resource control is then discussed.

3.1 Actors Model

Actors is a formal model for building and representing the behavior of concurrent objects, and is used to model computations in multi-agent systems. In his project PLANNER [38], Hewitt proposed Actor [39] as a unified formalization emphasizing the inseparability of control and data flow. In 1986, Gul A. Agha further developed Actors as a foundation for concurrent object-oriented programming [37].

An actor is an autonomous computing element which encapsulates a state, a number of procedures which can change the state, and a thread of control. Actors execute in parallel and communicate through asynchronous message passing. Each actor has a globally unique address and a message queue which serves as a buffer for its received messages. Figure 3.1 illustrates actors and message passing between actors.

Buffered messages are processed in their arriving order, and processing of each message causes a corresponding behavior of the actor. The Actors model defines three basic behaviors:

- creating new actors
- sending messages to actors whose addresses are known
- changing to a new state after processing a message

In this work, actors are used to model computations, and they are the elements which carry out communications and consume network resource.

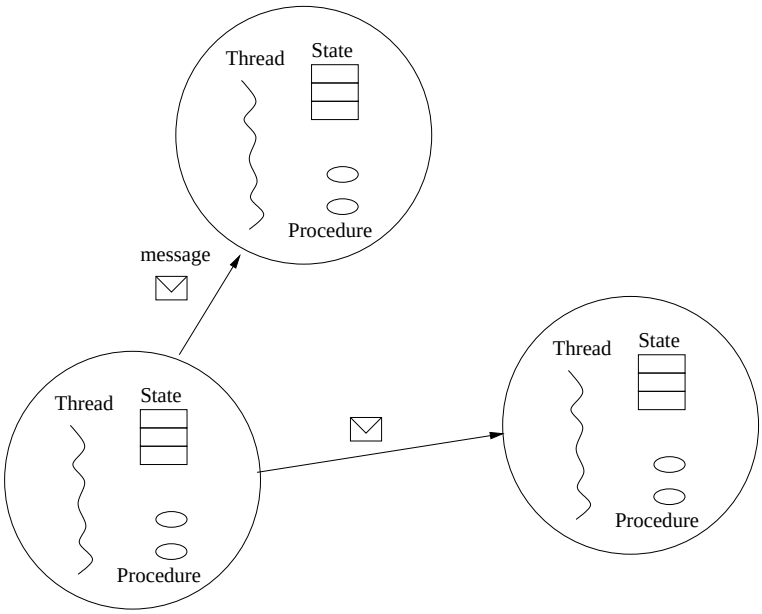

Figure 3.1: Actors Model

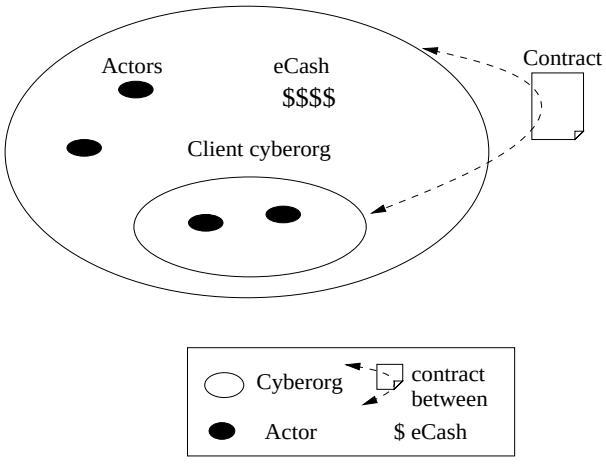

Figure 3.2: Cyberorg Composition

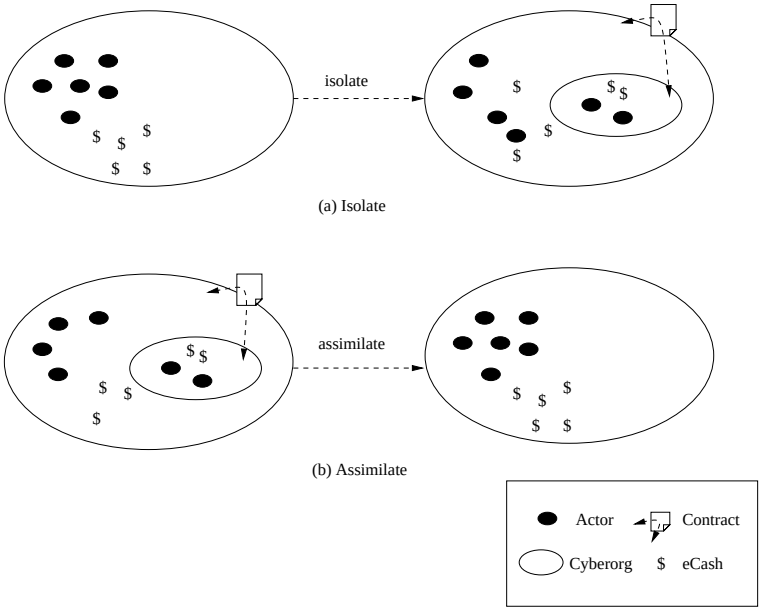

Figure 3.3: Isolate and Assimilate

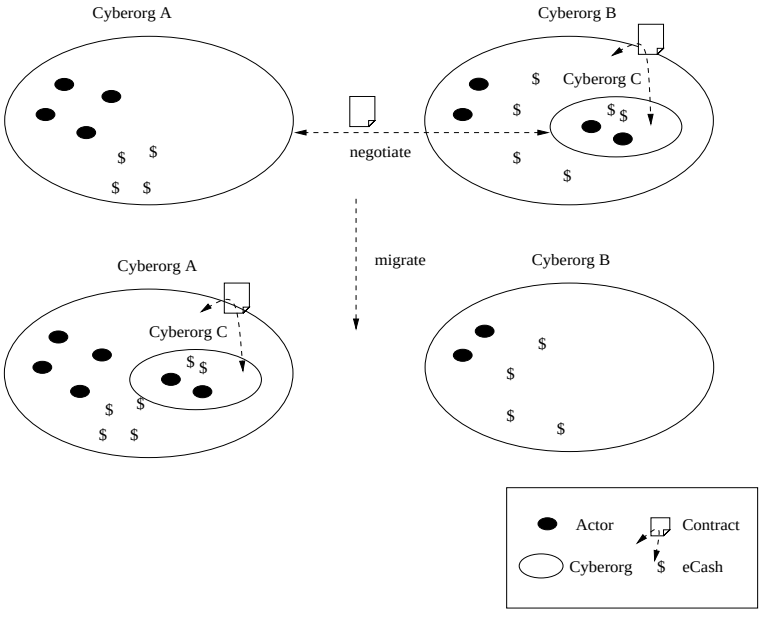

Figure 3.4: Negotiate and Migrate

3.2 CyberOrgs Model

CyberOrgs [2] is a model for resource management in multi-agent systems. The uniqueness of CyberOrgs is reflected in three aspects. First, in CyberOrgs, a computation and its sub-computations are bounded to a certain amount of resources. These computations cannot access resources external to this boundary. Second, CyberOrgs models ownership: resource has an owner at any instant of time, and the owner decides how its resources are used. Third, a market of resource is established in the CyberOrgs model, enabling trade in resource using eCash which serves as network currency.

A cyberorg is the basic unit of resource control, and cyberorgs are organized in a hierarchy. Figure 3.2 gives a snapshot of a cyberorg. A cyberorg encapsulates a number of contents, including actors, network resource, eCash and client cyberorgs. The root cyberorg conceptually owns all available resource in the world. As to other cyberorgs, each cyberorg receives resources from another cyberorg according to a siged contract. The containing cyberorg is called parent cyberorg to its contained cyberorgs. An actor carries out functional computation and consumes resources. ECash is the network currency which is used by cyberorgs to purchase resources from other cyberorgs. A cyberorg also hosts client cyberorgs and transfers resources to them in exchange of eCash according to the signed contract and thus forms a hierarchy.

The CyberOrg Model defines a number of primitive operations and three of them are focused including: isolate, assimilate and migrate.

- **Isolate** enables a cyberorg to spawn child cyberorgs. As illustrated in Figure 3.3, a new cyberorg is created inside the original one (parent cyberorg). The parent cyberorg transfers an amount of eCash, and a number of actors to the new cyberorg. In addition, a new contract is imposed on the child cyberorg, which stipulates the type and quantity of the resource received from the parent cyberorg, as well as the price the child cyberorg has to pay for the resource.

- **Assimilate**, on the contrary, allows a cyberorg to terminate and be absorbed by its parent cyberorg. All contents of the assimilated cyberorg (actors, resources and eCash) become contents of its parent cyberorg. Figure 3.3 shows this procedure.

- **Migrate** enables a cyberorg to move to another cyberorg for better resources after a successful negotiation of a contract. This procedure is illustrated in Figure 3.4.

3.3 Reification of Network Resource Control

CyberOrgs is a general model for resource management. In this work, the CyberOrgs model is specialized for controlling network resources. In this section, important aspects of this specialization

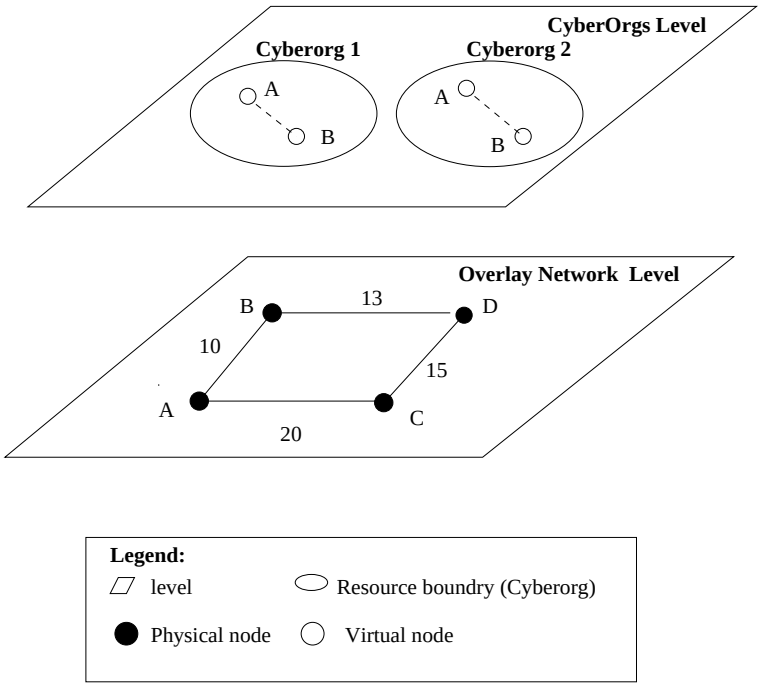

Figure 3.5: Virtual Link Mapping

are described.

3.3.1 Network Resource Abstraction

In order to control network resource, it is necessary to address how to model network resources. In this work, a *vLink* (Virtual Link) abstraction of network resource is introduced to characterize properties of a network connection (not necessarily direct) between two computer(or nodes). Using this abstraction, programmers are able to express their resource requests in terms of a set of attributes, rather than concrete configurations. These attributes can fall into spatial, temporal and QoS related. Spatial attributes include the source and destination nodes of a network connection, and the bandwidth (average data transfer rate) of the connection. Temporal attributes consist of availability state (either free or taken-up) and duration of a network connection. The attribute of duration is measured by the total amount of data allowed to transfer through the connection, and is called *data transfer limit*. QoS attributes may include average packet delay or packet loss rate of a connection.

By excluding concrete configuration of a network connection, vLink reflects the dynamicity of possible connections formed in a network and offers a simplified and uniform representation of network resource. With the vLink abstraction, low-level details of network resource are transparent to application programmers, and they are enabled to request network resources by quantifying vLink

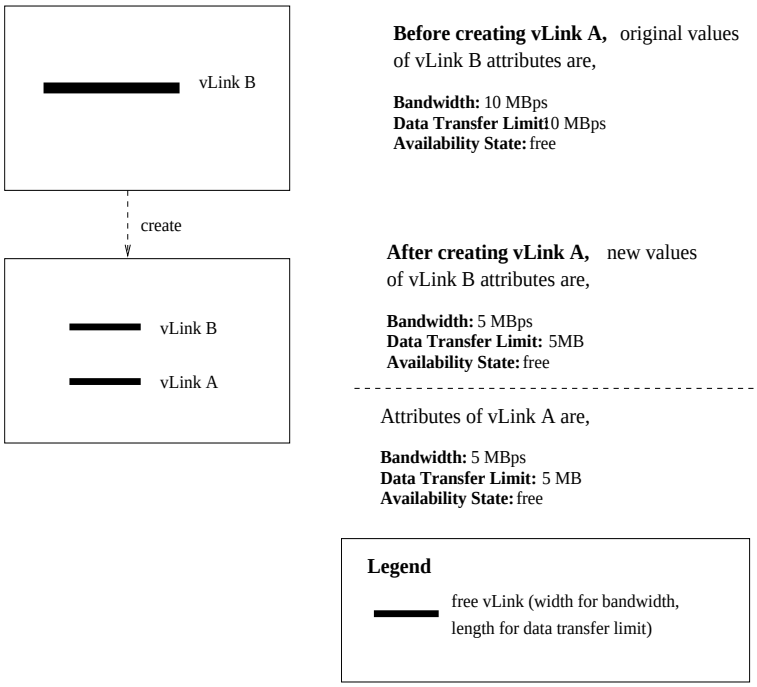

Figure 3.6: Vlink Creation (Partial Partition)

attributes.

'vLink' has different meanings when used in different contexts. First, theoretically, vLink is a class abstracting network resource and represents a complete set of possible network connections established in a space over time. Second, in the CyberOrgs system, a vLink is a concrete instance of virtual link which maps to a specific connection in the network. The term vLink or vLinks is applied in this case. Last, application programmers declare their network resource requests in the form of virtual link. We use vLink request for network resource request.

Figure 3.5 shows the mapping between vLinks in CyberOrgs and connections in network. A network connection is a path which goes through a number of nodes connected by overlay links. Here, a triple $Vlink(source,\ destination,\ bandwidth)$ is used to represent a vLink, and a $N+3$ tuple $Path(source,\ inter1,\ \ldots,\ interN,\ destination,\ bandwidth)$ is used to denote a network connection. N is the number of intermediate nodes. In Figure 3.5, Cyberorg1 holds a vLink, $Vlink(A,\ D,\ 10)$ which maps $Path(A,\ B,\ D,\ 10)$, and in Cyberorg2, $Vlink(A,\ D,\ 10)$ maps to $Path(A,\ C,\ D,\ 10)$. Although these two vLinks have different configurations, they are equivalent to programmers, as long as no other attributes are used to differentiate them.

Besides the relationship between vLinks and network connections, vLinks have relationship between themselves as well. A new vLink may be created from existing vLinks. There are two types of vLink creation, which are full-partition creation and partial-partition creation. As shown in Figure 3.6, vLink A is created from vLink B by partially occupying its bandwidth. After this

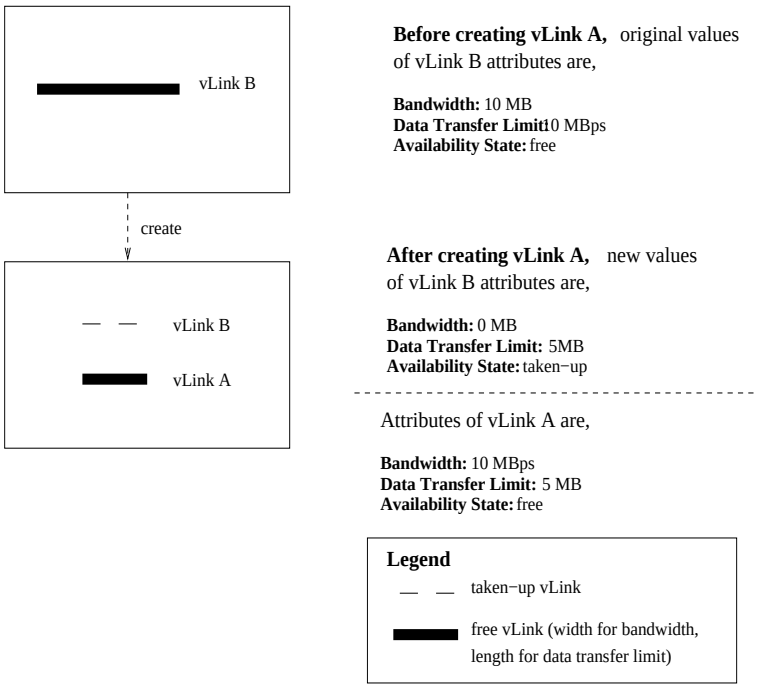

Figure 3.7: Vlink Creation (Full Partition)

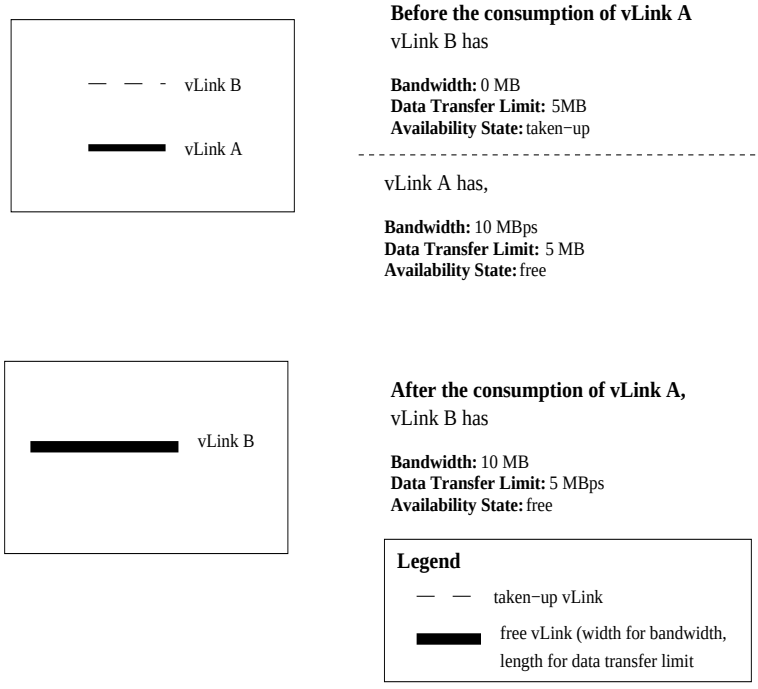

Figure 3.8: Vlink Consumption

creation, vLink A and vLink B are available at the same time and they share the space of the original vLink B. In this case, the availability state of vLink B is not influenced by the creation of vLink. By contrast, vLink A is created from vLink B by fully taking its bandwidth in a full-partition pattern creation (Figure 3.7). Thus, although vLink B may still have data transfer limit, the availability state of vLink B is changed to taken-up. In another word, vLink B is not available until the consumption of vLink A. A vLink is consumed when its total data transfer amount reaches its data transfer limit, as illustrated in Figure 3.8. Therefore, a partial partition pattern creation establishes a space-sharing relationship between two vLinks, and a full-partition pattern creation sets up a time-sharing relationship.

3.3.2 Internally Distributed Cyberorg

After the introduction of the vLink abstraction, a cyberorg for network resource control contains a number of actors and vLinks. An actor communicates with other actors within the cyberorg by consuming vLinks. Because a cyberorg can have vLinks on multiple nodes, a cyberorg containing vLinks is an internally distributed cyberorg.

3.3.3 System Cyberorg

The system cyberorg is the first cyberorg of the CyberOrgs system (the system is a close networked system). This cyberorg is special because it holds all resource of the system. The system cyberorg is not the root cyberorg, but is held by the root cyberorg, which conceptually contains all resources in the world. In the system, all other cyberorgs are isolated from the system cyberorg.

3.3.4 System Communication Cyberorg

In an internally distributed cyberorg, it is not only application actors that consume network resource for application-specific communication, but system actors (facilitators, described in Chapter 4) also consume network resources for system communications. In order to harmonize both application and system communications, a certain amount of resource has to be reserved for the latter. We introduce a special cyberorg, system communication cyberorg, which hosts all system actors and resources reserved for system communications.

Chapter 4

System Architecture Design

A prototype system is developed to reify network resource in multi-agent systems. This chapter describes the prototype system, which is a middleware system architecture used to build execution environment for network resource control. This execution environment is called a *CyberOrg platform*. A CyberOrg platform is composed of three layers, which are:

- *CyberOrgs interface layer* provides an application programming interface by which programmers are able to express resource requests and define resource control policies for their applications.

- *CyberOrgs management layer* enforces CyberOrgs mechanisms and transforms application-level resource requests into network-level resource allocation tasks.

- *Resource scheduling layer* enforces resource allocation tasks in the network.

Figure 4.1 shows the layered system architecture. This chapter describes design decisions made for each layer.

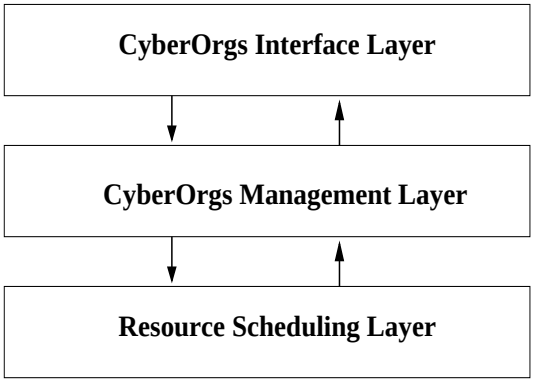

Figure 4.1: System Architecture

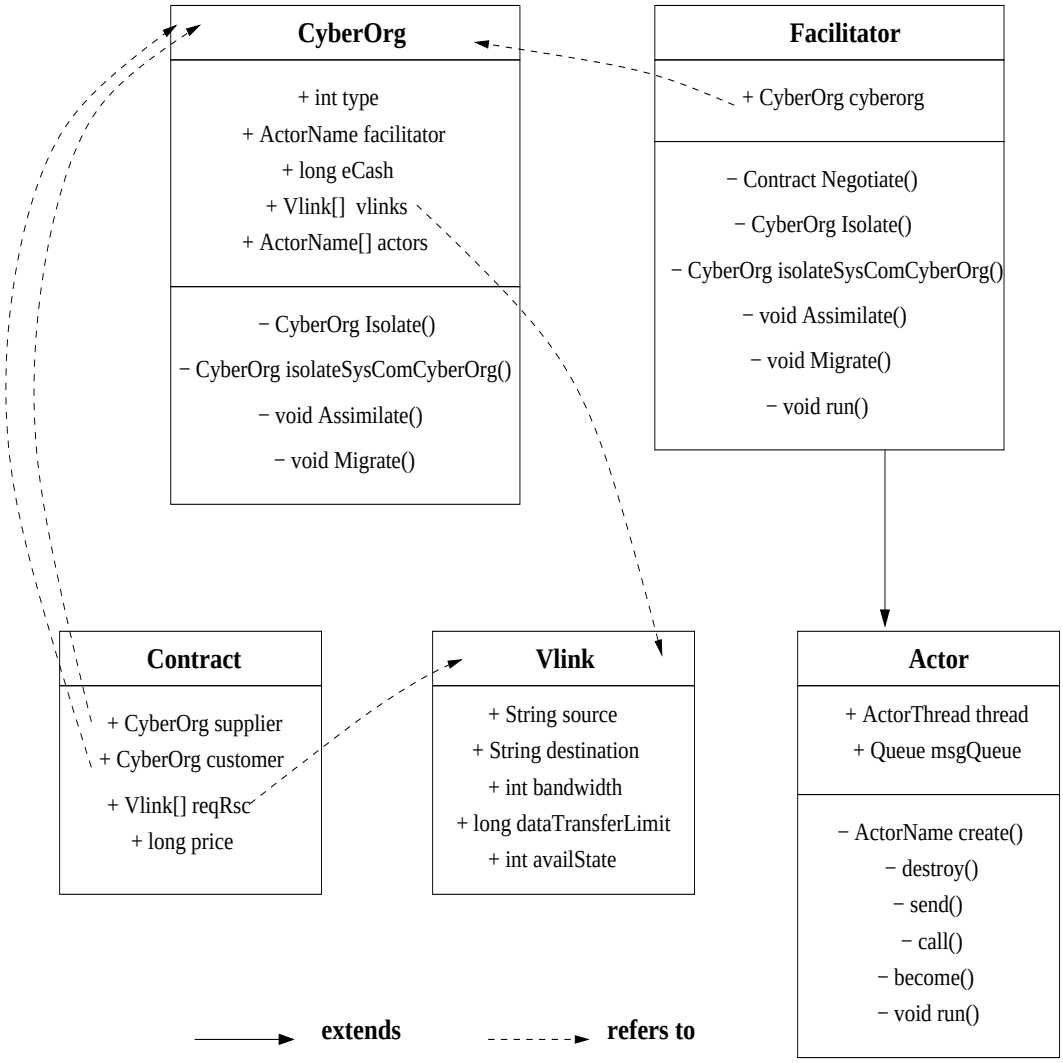

Figure 4.2: Class Relationships

4.1 CyberOrgs Interface Layer

CyberOrgs interface layer is composed of two parts: a class library and an application programming interface (API) for CyberOrgs. Using these facilities, programmers are able to specify application-level resource requests for their applications, and define their policies to control resource. In this section, the CyberOrg class library and APIs, and application examples are presented.

4.1.1 CyberOrgs Class Library

The CyberOrgs library contains a set of classes corresponding to major components in the model of CyberOrgs. Figure 4.2 shows the structure of the main classes which compose the CyberOrg

library. In the figure, there are five main classes: CyberOrg, Actor, Facilitator, Vlink and Contract. Functions of these classes are described in the rest of this section.

The CyberOrg Class A cyberorg is the basic unit of resource control in the CyberOrgs model. The facilitator field keeps the name of the facilitator which represents a cyberorg to control local resources. The other fields correspond to contents held by a cyberorg, including eCash, vLinks and actors. The methods which invoke CyberOrgs primitive operations are defined in this class as well. The *type* field is used to differentiate system cyberorg and system communication cyberog from application cyberorgs.

The Actor Class An actor in CyberOrgs carries out functional computations and consumes resources. The Actor class contains a thread and a message queue which buffers received messages. Besides providing the methods of actor creation, message passing and state changing, this class can also be extended by programmers to define application-specific methods.

A facilitator is a special actor, which facilitates controlling resource in a cyberorg. The Facilitator class extends the class of Actor, and contains a reference of a cyberorg. This class contains methods to negotiate with other cyberorgs, and methods for corresponding primitive operations. Every request for resource or primitive operations eventually arrives at the corresponding facilitator of a cyberorg as a message. When a facilitator is started, the run() method (shown in Figure 4.2)executes a main loop which processes these messages by executing primitive operations.

The Facilitator class can be extended for two main purposes. First, programmers extend this class to develop their own resource control policies. An example is shown in Section 4.1.3. Furthermore, an internally distributed cyberorg may have a number of facilitators working coordinately. Programmers are not constrained in the choice of interaction protocols for facilitators. For example, they may either use a master-slave pattern protocol for facilitators to coordinate, or apply a peer-to-peer interaction protocol.

The Vlink and Contract Classes Vlink is the class for vLink. In this class five fields are defined, and each corresponds to an attribute of the vLink. The fields of *source* and *destination* are used to specify the source and destination nodes of a vLink, and the *bandwidth* field specifies the amount of bandwidth of the vLink. The fields of *availState* and *dataTransferLimit* are for specifying the availability state and duration of a vLink.

In the Contract class, the *supplier* and *consumer* fields match to the cyberorg that grants and receives resources respectively. The requested resource and corresponding price are specified in the fields of *reqRsc* and *price*.

4.1.2 CyberOrgs APIs

In this section, APIs for Actors and CyberOrgs are presented.

Actor APIs Most Actor APIs provided by Actor Architecture [40], which is a middleware architecture for building Actor systems, are used in the prototype system, except for the actor creation. Two new methods are added for creating actors.

- Actor Creation
  - ActorName createActor(ActorName p_anCreator, String p_strActorClass, Object[] p_objaArgs)
    This method is called to create an application actor. Parameters include: (a) p_anCreator indicates which actor creates the new actor; (b) p_strActorClass is class name for the new actor, and (c) p_objaArgs are the arguments used to create the new actor.
  - ActorName createActor(CyberOrg p_coMyCyberOrg, String p_strFacilitatorClass, Object[] p_objaArgs)
    This method is invoked by the cyberorg constructor to create a facilitator actor. Different from the above application actor creation, this method has a parameter of p_coCyberOrg which indicates which cyberorg the facilitator represents.

CyberOrg APIs We define the APIs for Cyberorg creation, absorption, negotiation and migration.

- Cyberorg Creation
  - Cyberorg creatCyberOrg(String p_strCyberOrgClass, long p_lECash, Contract p_cContract, String p_strFacilitatorClass, Object[] p_objaArgs)
    This method is called to create a cyberorg, which can be the system cyberorg or an isolated cyberorg. In the latter case, is method is invoked by the Isolate() method, which will be introduced later. Parameters include: (a) p_strCyberOrgClass is the class name of the new cyberorg; (b) p_lECash is the amount of eCash provided to the new cyberorg; (c) p_strFacilitatorClass is the class name of the facilitator; (d) p_cContract is the contract imposed on the new cyberorg[1], which specifies the network resource given to the new cyberorg, and the price of the resource, and (e) p_objaArguments are the arguments needed to create the facilitator.
  - CyberOrg isolateSysComCyberOrg(long p_lECash, String p_strFacilitatorClass, Contract p_cSysComContract, String p_strFacilitatorClass, Object[] p_objaArgs)

[1] If the cyberorg is a system cyberorg, the contract is between this cyberorg and the system. Otherwise, the contract is between this cyberorg and its parent cyberorg.

This method is called to create a system communication cyberorg. It has similar parameters as the createCyberOrg method, except that p_cSysComContract is the contract between the system cyberorg and the system communication cyberorg, which indicates the network resource reserved for system communications, and the price of the resource as well.

  - Cyberorg Isolate(long p_lECash, ActorName[] p_anaActors, Contract p_cNewContract)
    This method is called to create a new child cyberorg. Parameters include: (a) p_lECash is the amount of eCash given to the new cyberorg; (b) p_anaActors is an array of existing actors which are isolated into the new cyberorg, and (c) p_cNewContract is the contract between the new cyberorg and the host cyberorg, which denotes the resource granted to the new cyberorg, as well as the cost of the resource.

- Cyberorg Absorption
  - void Assimilate()
    This method is invoked to assimilate a cyberorg. After the invocation of this method, all the resources, actors and eCash held by the assimilating cyberorg are released to the host cyberorg.

- Cyberorg Negotiation and Migration
  - Contract Negotiate(ActorName p_anSupplierFacActor, Contract p_cProposedContract)
    This method is called by a customer[2] cyberorg to initiate a negotiation with another cyberorg in order to purchase resources. Parameters include: (a) p_anSupplierFacActor is the name of the facilitator of the supplier cyberorg, and (b) p_cProposedContract is the proposed contract which specifies the customer cyberorg's resource request and the proposed price. If the supplier cyberorg has the requested resource and agrees to offer it, a contract with the offered resource and its price is returned.
  - void Migrate(ActorName p_anDestinationActor, Contract p_cNewContract)
    This method is called to migrate a cyberorg to the destination cyberorg. Parameters include: (a) p_anfacActorOfdesCyberorg is the name of the facilitator in the destination cyberorg, and (b) p_cNewContract is the contract between the migrating cyberorg and the destination cyberorg, which is generated by negotiation.

4.1.3 Chat Room Service Example

In this section, a chat room service example is given to illustrate how to develop applications using APIs of the system. In this example, it is assumed chat room service provider offers multi-type

[2]The cyberorg which initiates a negotiation.

services such as online conference, chatting, file transferring, audio and video transferring and so on. For commercial benefits, the service provider needs to offer differentiated services to different users. Furthermore, the service provider also concerns issues of system stability and security, therefore resource control is required by the service provider. In the remaining part of this section, this example is used to show: (1) how to set up an application in the system; (2) how to develop a resource allocation policy and (3) how to develop application-specific resource control policies. In this example, it is assumed that a chat service library to support necessary chat room functions and relevant utilities already exist. In addition, it is assumed that the CyberOrgs system is assumed to be already installed and running on a dedicated network.

Chat Room Service Setup

In order to isolate a new cybeorg for this chat room service, the name of the master facilitator of the system cyberorg has to be known in advance. This is done by retrieving the actor name of the master facilitator of the system cyberorg from the local CyberOrg Manager. The CyberOrg Manager, which manages local cyberorg hierarchy, has a public name (uan://host_address:2). Then, the resource allocation specification file (described in the next section) is loaded to set default allocation policy held by the new cyberorg. After a contract is generated for this isolation, a synchronous message is sent to the master facilitator of the system cyberorg to isolate a cyberorg for this chat room service.

Resource Allocation Specification

In order to offer different qualities of service to users, the service provider has to specify an allocation policy. There are different possible ways to achieve this purpose (e.g. using a resource specification language such as RSL provided by Globus [15])). Here a simple example is given of a specification file.

It is assumed that the service provider allocates vLinks with different bandwidth and data transfer limit to different services. Therefore users of the same service are allocated with equalized vLinks. Figure 4.4(a) shows an allocation specification file in which resource allocation policy is in terms of the service type, such as chat service (represented by the corresponding actor class), the amount of bandwidth required, and the data transfer limit. Alternatively, bandwidth requirement can be specified qualitatively as shown in Figure 4.4 (b).

Each cyberorg has such a specification file, and allocates resources to actors according to the specified policy.

Resource Control Policy

Although a resource allocation specification file allows service providers to set default allocation policy for their resources, this is only one side of the story. Service providers may also want to

```
public void createChatRoomService()
{
      // 1. Retrieves the master facilitator
      //    name of the system cyberorg
      ActorName anCyberOrgManager = CyberOrgManager.getName()
      ActorName anSystemMasterFacilitator = call(anCyberOrgManager,
                                 getSystemCyberOrg)

      // 2. Creates a contract which contains
      //    resource request
      Contract cNewContract =
         ContractManager.generateIsolationContract(
                     anSystemMasterFacilitator, vLinkRequests);

      // 3. Sets up control policy
      BasicAllocationPolicy bapPolicy = new BasicAllocationPolicy();
      bapPolicy.loadDefaultAllocation (specificationFile);

      // 4. Sends a synchronous isolate request
      call(anSystemMasterFacilitator, Isolate,
           100, m_anMyName, cNewContract, bapPolicy)

      // 5. Starts the service
}
```

Figure 4.3: Chat Room Service Setup

Service	Bandwidth Request	Data Transfer Limit
ChatActor	0.1MBps	50MB
ConferenceMemberActor	1MBps	100MB
FileShareActor	0.05MBps	10MB
VideoShareActor	10MBps	500MB

(a) Resource Request by Quantity

Service	Bandwidth Request	Data Transfer Limit
ChatActor	fast	50MB
ConferenceMemberActor	faster	100MB
FileShareActor	normal	10MB
VideoShareActor	fastest	500MB

(b) Resource Request by Quality

Figure 4.4: Sampel Resource Specification File

change the default allocation policy to meet some special requests. For example, an application user wants to have faster service for file transfer service users. In addition, sometimes, such requests may be so aggressive that contradict to service providers' concerns, or even cause safety issues, and need to be rejected. With these concerns, application programmers should define policies about how to control resource allocation. In Figure 4.5, the pseudo code of a resource control policy is given. This method is defined in a Chat Facilitator class. In this example, a threshold value is set to restrict the amount of resource requested by the user. If a request is greater than the threshold, it will be refused. If the resource request can not be satisfied, this facilitator will negotiate with a potential supplier which has better resource and initiate a migration. Furthermore, if the number of actors in a cyberorg is greater than 10, the isolate operation is invoked and a new cyberorg will be created. When there are no application actors in a cyberorg, the cyberorg assimilates.

4.2 CyberOrgs Management Layer

CyberOrgs management layer enforces resource management mechanisms of CyberOrgs. This layer is designed for supporting CyberOrgs hierarchy management, resource allocation and contract management. The structure of CyberOrgs management layer is shown in the dashed frame in Figure 4.6. In this layer, three components are involved which are the CyberOrg Manager, Task Distributor and Contract Manager. CyberOrg Manager is the module which performs CyberOrgs hierarchy

```
public void resourceAllocation()
{
      if(vLinkRequest.getBandwidth() > threshold){
                  // refuse this request;
      }else{
            boolean bSuccess = Allocate(vLink);
            if(bSuccess ==  false) {
                  ActorName potentialSupplier =
                        lookupYellowPage(vLinkRequest);
                  Contract migrationContract = generateMigrationContract(
                        customer, vLinkRequest, proposedPrice);
                  Contract signedContract = Negotiate(potentialSupplier,
                                    migrationContract);
                  if(signedContract == null){
                        // looks up another potential supplier and
                        // initiates a new negotiation
                  }else{
                        Migrate(potentialSupplier, signedContract);
                  }
            }

            if(actorList.size() > 10){
                  Contract newContract =
                        ContractManager.generateIsolationContract(
                                    supplier, vLinks);
                  // decides actors, eCash to be isolated
                  Isolate(eCash, actors, newContract);
            }else if(actorList.size() <= 1){
                  Assimilate();
            }
      }
}
```

Figure 4.5: Sample Local Resource Control Policy

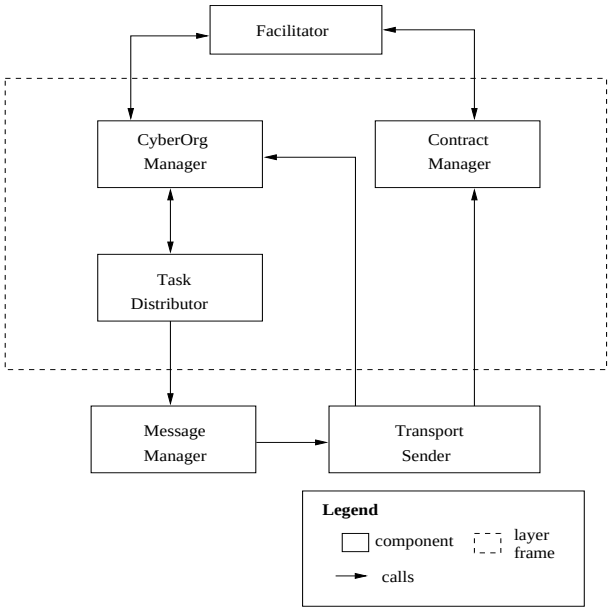

Figure 4.6: CyberOrgs Management Layer Structure

management, and cooperates with the Task Distributor to carry out resource allocation. A Contract Manager is in charge of guaranteeing the execution of contracts made between cyberorgs.

In this part, the major services supported in the layer are described. These services are CyberOrgs hierarchy management, vLink allocation and contract management.

4.2.1 CyberOrgs Hierarchy Management

Resource control in CyberOrgs is organized in a hierarchical way, and this control structure is changed through CyberOrgs primitive operations. For example, when the Isolate primitive is invoked, a cyberorg creates a new cyberorg and gives a set of vLinks to the child cyberorg to control. A CyberOrg Manager maintains the hierarchical structure of the cyberorgs created on a CyberOrg platform. When a primitive operation is invoked, the CyberOrg manager modifies the hierarchy to reflect the change in the structure. On each cyberorg platform, there is one cyberorg manager which performs local cyberorg hierarchy management.

4.2.2 Resource Allocation

Resource allocation is the core mechanism in the CyberOrgs system. As vLink is the singular form of network resource in CyberOrgs, the resource allocation has different interpretations at the CyberOrgs level and network level. At the CyberOrgs level, resource allocation is to create vLinks upon requests and assign these vLinks to the requester, either a cyberorg or a communication. In addition, due to the duration property of a vLink, some post-allocation maintenance work has to be

performed after a vLink is consumed. Correspondingly, at the network level, resource allocation is to search a path to meet the source, destination and bandwidth requirements specified in a resource request, establish a connection along the path for communication, and disestablish this connection at the end of the corresponding communication. The resource allocation enforcement is achieved by resource scheduling layer (see next section). We call the procedure of vLink creation and path searching as *resource discovery*, and vLink assignment and connection establishment as 'resource allocation task distribution'. The procedure of vLink consumption and connection disestablishment is called *post-allocation maintenance*.

Algorithm 1 Customized Dijkstra(G, w, s, t) (G is the network graph containing a set of vertexes V and edges E; w is the weight of a direction connection between node x and y; s is the source node; t is the destination node)

```
1: N ← {s}
2: for each node v ≠ s do
3:     C(v) ← w(s, v)
4: end for
5: maxCapacity ← 0
6: maxCapacityNode ← NIL
7: while N does not contain all vertices do
8:     for each node i ∈ G and i ∉ N do
9:         if C(i) > maxCapacity then
10:            maxCapacity ← C(i)
11:            maxCapacityNode ← i
12:        end if
13:    end for
14:    N ← N ∪ maxCapacityNode
15:    for each node j ∉ N do
16:        if C(i) < min(C(maxCapacityNode), w(maxCapacityNode, j)) then
17:            d(j).enqueue(maxCapacityNode)
18:            C(j) ← min(C(maxCapacityNode), w(maxCapacityNode, j))
19:        end if
20:    end for
21: end while
22: return d(t)
```

Resource discovery

The procedure of resource discovery plays two important roles. First of all, this procedure controls

the admission of resource requests. If a resource request can not be satisfied, it will be refused. Further, this procedure maps an application-level vLink to a low-level network connection.

Upon receiving a vLink request, a concrete physical path which meets the source, destination and bandwidth requirements specified in the request has to be found before the vLink creation. This is in fact a routing problem. Different from routing on the Internet, which aims to find the shortest path between a source and destination, this routing also concerns bandwidth requirement. It is necessary to ensure that the searched path has bandwidth greater than or at least equal to the requested value.

With this consideration, a customized Dijkstra algorithm is developed. Algorithm 1 shows the algorithm. In this algorithm, G is a network graph which contains the sets of vertexes V and edges E. If x and y are two vertexes in V, $w(x,\ y)$ is the weight of the direct connection between x and y. The value of $w(x, y)$ is assigned to be zero if there is no direct connection between x and y. Otherwise, $w(x, y)$ is determined by the bandwidth of the connection between x and y. At each vertex, the path with maximum weight starting from the vertex itself (marked as s) to every other vertex in the network is computed. $C(v)$ is used to save the capacity of the path from s to vertex v, and $d(v)$ is used to keep the path from s to v. N is an array used to keep all vertices in the network, and only contains s initially. t represents the sink vertex.

Resource Allocation Task Distribution

After finding a required path, a corresponding vLink is created to represent the resource at the CyberOrg level. At the low level, a connection has to be established. Each node along the path has to be informed about their scheduling tasks corresponding to the resource request. The Task Distributor is responsible for distributing a resource allocation task to related nodes. Figure 4.7 shows this procedure.

In Figure 4.7, the CyberOrg Manager passes a resource allocation task to the Task Distributor. A resource allocation task stipulates which flow a vLink is created for, the underlying path mapping to this vLink as well as the bandwidth of the vLink. Each node along the path has to schedule packets belong to the flow according to the allocated bandwidth. There are two ways for a Task Distributor to inform related nodes about the task. First, the Task Distributor generates a single control message which contains the resource allocation task, and relays this message along the assigned path. Each node along the path receives the message and sets a state in itself. It is the last node who receives the control message that sends back a response message to the Task Distributor. Then, the Task Distributor informs the Message Manager to start the corresponding communication which requires this resource allocation[3]. Another way is to generate a control message for each node involved and send the message directly to each node. In this way, the Task

[3]The procedure of starting a communication is message multiplexing procedure described in Section 4.3

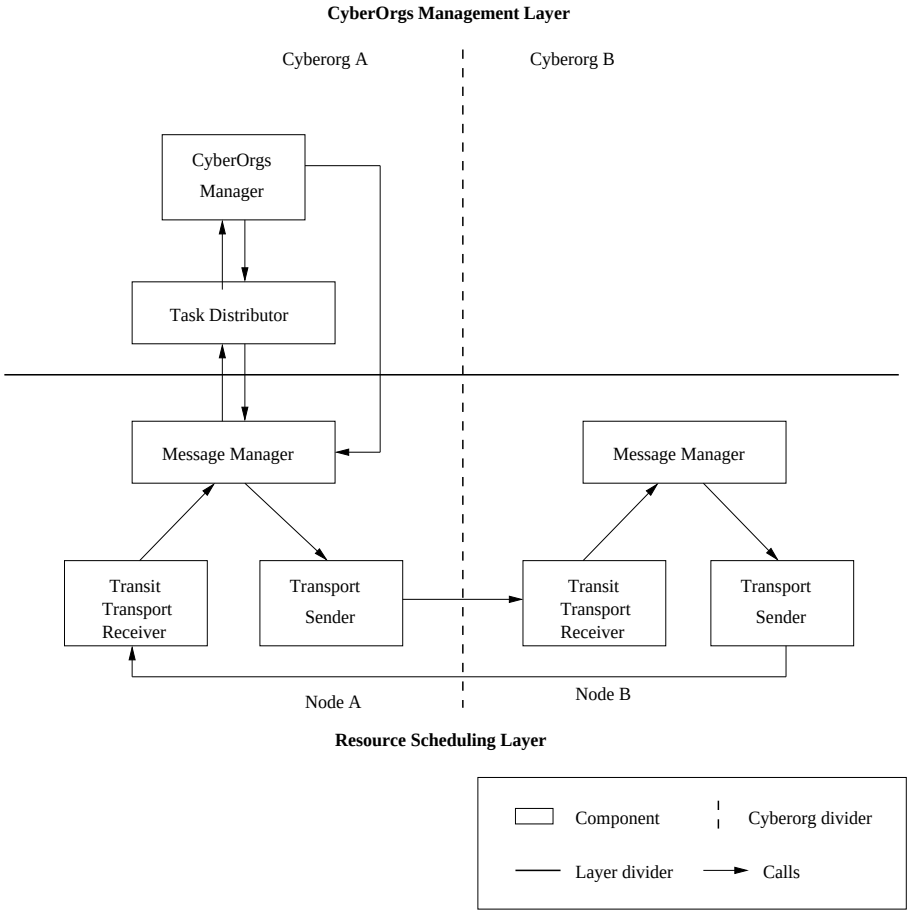

Figure 4.7: Task Distribution Procedure

Distributor has to wait for response messages from all nodes. This way is not a good solution because more messages are generated which increases system traffic.

Post-Allocation Maintenance

Due to the duration attribute of vLink, a vLink has a lifespan from its creation to consumption. Accordingly, when a vLink is consumed, the underlying connection should be disestablished as well. This procedure is accomplished by Transport Sender in the third layer. A Transport Sender schedules packets for each communication according to their requested bandwidth. It is this module that is able to know when the data transferred through a vLink reaches its limit. Then the Transport Sender relays a disestablishment message to all nodes involved in the path of the connection. Allocation task corresponding to this vLink is deleted at each involved node.

4.2.3 Contract Management

In CyberOrgs, vLinks can be transferred from one cyberorg to another according to the contract agreed between them. After a contract is signed between two cyberorgs, a buyer and seller relationship is established. According to the contract, the seller has the responsibility to provide vLinks requested by the buyer, and the buyer has to pay eCash for the vLinks. In order to guarantee contract fulfillment, contracts have to be kept and enforced by a third party representing the system. Contract Manager is such a component which has the privilege given by the system to enforce contracts. In the system, the sellers responsibility of providing requested vLinks are guaranteed by the resource allocation mechanism, Contract Manager is responsible for charging the payment from the buyer. In order to achieve this, each cyberorg has a bank account and the Contract Manager can access to these accounts and charge the payment.

4.3 Resource Scheduling Layer

So far, the design of system interface and CyberOrgs mechanism enforcement has been discussed. In this section, the design of the resource scheduling layer which implements the Actors model and supports CyberOrgs management layer by enforcing resource allocation in the network is described. Three types of scheduling are built in this layer to achieve flow multiplexing, bandwidth control, and CPU time allocation.

4.3.1 Actor Architecture Customization

Resource scheduling layer is implemented by modifying an existing Actor system, Actor Architecture [40] (AA). AA has a well-designed layer architecture, which is easy to extend. Most of

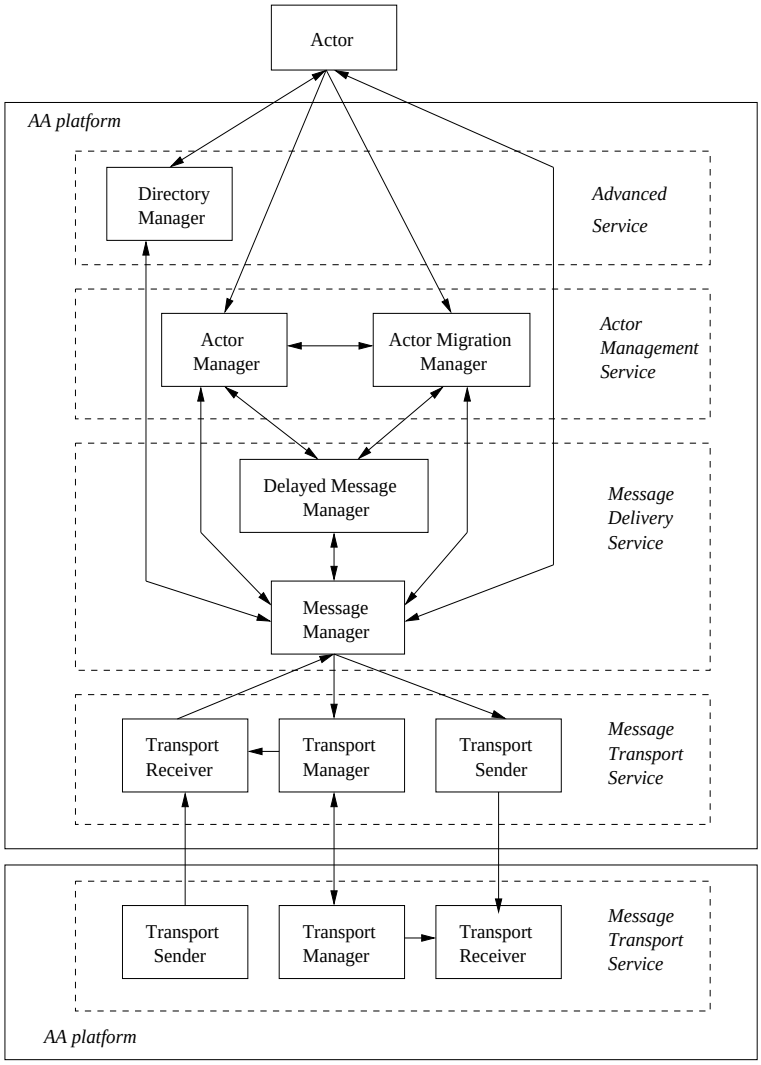

Figure 4.8: Actor Architecture

the original components in AA are retained and some are customized to support the proposed scheduling schemes.

Actor Architecture

Actor Architecture [40] is middleware system architecture to build Actor systems. Figure 4.8 gives the conceptual architecture of AA. Actors developed by programmers execute on an AA platform, which is an actor execution environment running on a computer node. An AA platform is composed of four service layers, which are:

- Advanced Service provides middle actor service, such as matchmaking and brokering services. Currently it only has one component, the Directory Manager.

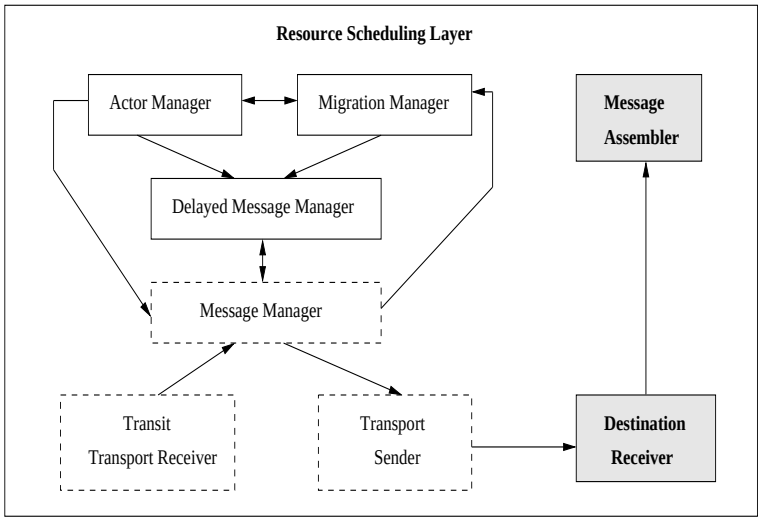

Figure 4.9: Customized Actor Architecture

- Actor Management is responsible for managing the states of all actors on the platform.
  - The Actor Manager manages the states of actors running on the node and mobile actors
  - The Actor Migration Manager manages actor migration.
- Message Delivery Service handles all the local messages delivery in the AA platform.
  - The Message Manager handles messages passing among actors. It delivers messages destined to the same AA platform directly. For messages, targeting to other nodes, MM send them to its counterpart of the AA platform on the destination node.
  - The Delayed Message Manager temporally holds messages for mobile actors when they are moving from their AA platforms to other AA platforms.
- Message Transport Service is in charge of communications with other AA platforms.
  - The Transport Manager maintains a public port and sets up TCP connections with other AA platforms.
  - The Transport Sender sends messages to other AA platforms, and contact with the Transport Managers of the destination platforms, in case that there are no established connections
  - The Transport Receiver is created after a new connection is established and receives messages from the other platform.

Customized Actor Architecture

Although AA provides a good architecture for building Actor systems, network resource control

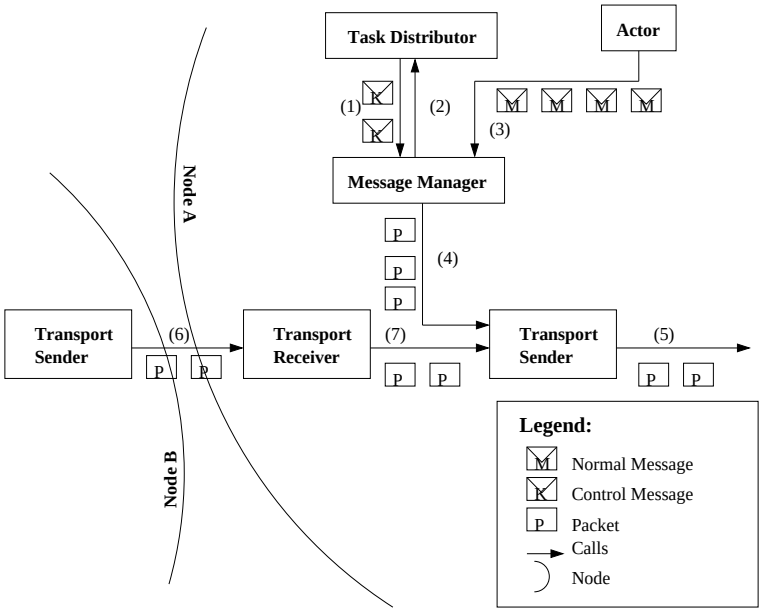

Figure 4.10: Source Message and Packet in Transmission Scheduling

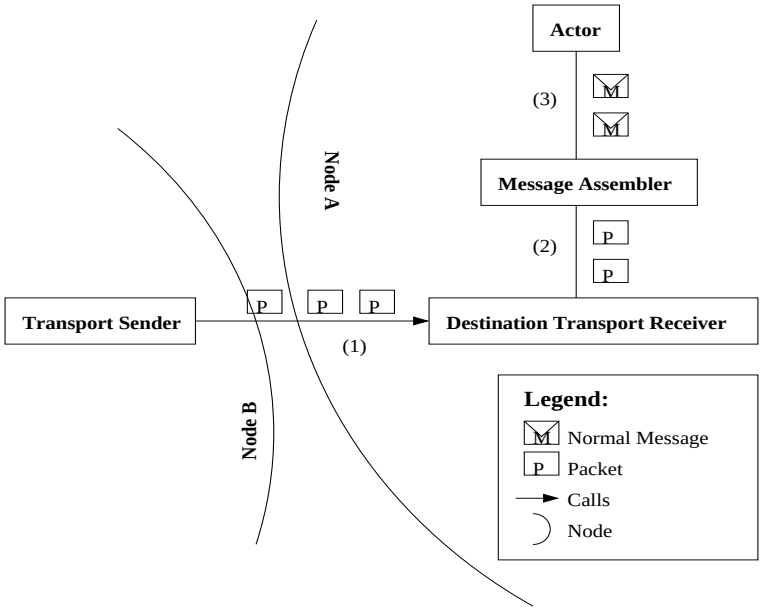

Figure 4.11: Destined Packet Processing

is not handled in the architecture, but is left to the network. Besides, since the Actors model only guarantees weak fairness [37], messages are ensured to be delivered eventually, but there is no guarantee about when they are sent. All messages are buffered and sent according to their arrival order. For the purpose of enforcing bandwidth control mechanisms, a few basic changes are made to AA communication services. First of all, UDP is used instead of TCP for communication. Further, schemes of message scheduling, packet scheduling, and thread scheduling are added to AA.

In Figure 4.9, modified components are labeled in dashed frames, and new components are in shaded frames. A message scheduling scheme is added to the component of Message Manager, so that a Message Manager is not only responsible for delivering messages, but also controls the order of processing messages. A Transport Sender is in charge of packet scheduling, which makes sure packets are sent at their requested rate. Transport Manager is removed because TCP connections do not need to be set up for different communications. Transport Receiver is split into two components: Transit Transport Receiver and Destination Transport Receiver. A Transit Transport Receiver is responsible for forwarding packets in transit to successive node, and a Destination Transport Receiver receives destined packets and passes them to a Message Assembler, which assembles packets belonging to the same flow into a message and forwards the message to the target actor.

Control flows are described in Figure 4.10 and Figure 4.11. There are three types of data to process, source messages, packets in transit, and destined packets. Source messages are the ones generated by local actors at a node. Messages are first scheduled by a Message Manager. After fragmenting these messages into packets, the Message Manager passes these fragmented packets to the Transport Sender, which schedules the order and rate of sending packets. This procedure is identified by (3) (4) (5) in Figure 4.10; Packets in transit are the ones which pass through a node, and need to be scheduled and forwarded to the next hop (illustrated by (6) (7) (5) in Figure 4.10); As to packet which are destined to a node, they are received by a Destination Transport Receiver and assembled into messages by a Message Assembler. Figure 4.11 shows this procedure.

4.3.2 CyberOrgs Scheduling

In the layer of resource scheduling, messages, packets and threads are scheduled. Message scheduling supports link sharing between different flows. Packet scheduling realizes rate-based bandwidth control, and thread scheduling guarantees any thread at a node has a chance to progress. With these three scheduling schemes, bandwidth allocation can be achieved in a well-behaved system. In order to have an efficient implementation, a flat scheduling approach described in [41] is employed. Although the hierarchical structure of CyberOrgs provides scalability, it also poses a challenge on efficient scheduling. Since the implementation of a hierarchical coordination is very costly [42], scheduling is performed at each node without awareness of CyberOrgs.

Algorithm 2 Message Scheduling(Tgranularity) ($Tgranularity$ is the time granularity of each scheduling cycle)

```
1: while true do
2:     for each message queue i do
3:         if i is not empty then
4:             get the first message in the queue
5:             retrieve the stop position where last message decomposition stops
6:             retrieve allocated bandwidth B_i
7:             decompose N_i packets from this message /*N_i = B_i * Tgranularity*/
8:             enqueue these to packet queue i
9:             remember the stop position
10:        end if
11:    end for
12: end while
```

Message Scheduling

In this system, an actor communicates with other actors through sending asynchronous messages which may have different sizes. We decompose a large message into a number of fixed length packets (1024 bytes), and control the sending rate and order of these packets to enforce bandwidth allocation.[4] Although packets are scheduled at a rate corresponding to allocated bandwidth, it is still necessary to schedule messages for the purpose of multiplexing the link utilization.

Algorithm 2 shows the message scheduling algorithm, in which Tgranularity represents the length of each scheduling cycles. For each communication (message passing between two actors), there is a corresponding message queue to buffer messages for this communication, and a packet queue which buffers decomposed packets accordingly. Message decomposing and dequeueing happen in a round robin way. For example, at each message scheduling cycle, the first message in each message queue is allowed to decompose 20 packets and these packets are dequeued from the message queue and then put into the corresponding packet queue. The number of packets is the product depends on the bandwidth allocated to the communication and scheduling time granularity.

Packet Scheduling

The packet level scheduling is based on the weighted round robin algorithm, and also supports adjustable time granularity. The reason of choosing weighted round robin is because it enforces absolute bandwidth sharing between a number of flows and packets under scheduling have fixed size. By granularity, it is meant the length of each scheduling cycle. In this scheduling algorithm,

[4]The system has a required minimum length of incoming message. If a message is too small, i.e. 1024 bytes, the system will add extra bytes to it in order to enforce the requested bandwidth.

Algorithm 3 Packet Scheduling(Tgranularity) ($Tgranularity$ is the time granularity of each scheduling cycle)

```
1: while true do
2:     start timing /*start time is Tsend*/
3:     calculate the end time /* Tend = Tstart + Tgranularity*/
4:     for each packet queue i do
5:         if thread j is not empty then
6:             retrieve allocated bandwidth B_i
7:             dequeue N_i packets /* N_i = B_i * Tgranularity*/
8:             send these packets to the next hop
9:         end if
10:    end for
11:    message scheduling
12:    if currenttime < endtime then
13:        wait until the end time
14:    end if
15: end while
```

the weight of a flow is determined by its bandwidth request. When the granularity of a time cycle is set, the number of packets to be scheduled for a flow is the product of its weight and time granularity. Algorithm 3 shows the proposed packet scheduling scheme in this work. In this algorithm, $Tgranularity$ is the adjustable parameter. We use Tstart and Tend to denote start and end time of a scheduling cycle respectively.

Thread Scheduling

There are two reasons for having thread scheduling in the implementation. First of all, processor time is finite resource for which threads compete with each other. In the system, both system threads and application threads (actors) exist on each node. Without thread scheduling, some threads may dominate the processor, causing others having less chance to progress. In addition, packet scheduling is time sensitive, and it is necessary to ensure that during each time cycle, the required number of packets can be sent out in order to enforce bandwidth allocation. Because the prototype system is implemented as a multi-thread system, if thread scheduling is left to the operating system, there will be no guarantee that execution of other threads will not influence the packet scheduling. Motivated by these two reasons, the thread scheduling is combined with packet scheduling, and termed here system scheduling. The system scheduling scheme guarantees that: (1) all threads have opportunities to progress, and (2) in each scheduling cycle, packet scheduling is not influenced by other threads.

Algorithm 4 System Scheduling(Tgranularity) ($Tgranularity$ is the time granularity of each scheduling cycle)

```
1: initialize each system thread and register them in the thread queue
2: start all threads and suspend them right after starting them
3: while true do
4:     start timing
5:     calculate the end time /*Tend = Tstart + Tguranularity*/
6:     for each packet queuei do
7:         if i is not empty then
8:             retrieve allocated bandwidth B_i
9:             dequeue N_i packets /* N_i = B_i * Tgranularity*/
10:            send these packets to the next hop
11:        end if
12:    end for
13:    message scheduling
14:    compute thread execution time
       /*Texectime = (Tend − Tcurrent)/lengthOf(ThreadQueue)*/
15:    for each thread j in queue do
16:        if i is still alive then
17:            resume j
18:            wait Texectime
19:            suspend j and put j to the end of the thread queue
20:        else
21:            remove j
22:        end if
23:    end for
24: end while
```

In the scheme of system scheduling, there is one System Scheduler on each node responsible for scheduling all local threads. The System Scheduler has a thread queue. When a thread is created, it is suspended and put into the end of the thread queue. The System Scheduler wakes up the thread at the front of the queue, and schedules it to execute for a fixed time. When the time expires, the current running thread is put back to the end of the queue waiting for its next round. In each scheduling cyberorg, System Scheduler first executes packet and message scheduling, and then divides the rest time of each cycle among the other threads in the queue. This scheduling scheme is described in Algorithm 4.

Chapter 5

System Implementation

System design of the prototype system is presented in Chapter 4. This chapter describes implementation details of actor and cyberorg creation, primitive operations and resource allocation mechanisms, as well as computation and communication overhead analysis for CyberOrgs primitive operations.

5.1 Resource Allocation

As a core mechanism, resource allocation is carefully implemented in the system. In the previous chapter, resource allocation is divided into three parts: resource discovery, resource allocation task distribution and post-allocation maintenance. Implementation details of these procedures are presented as follows.

5.1.1 Resource Discovery

In Section 4.2.2, resource discovery is defined as the vLink creation in a cyberorg, and path searching in an overlay network.

Path Searching

Although path searching was discussed in an overlay network in Chapter 4, it can be applied at the CyberOrgs level as well. Because Virtual Link is the singular form of network resource in CyberOrgs, the CyberOrgs system can be viewed as a network composed of vLinks, and is called *vLink network*. A vLink request requires searching a 'path' in the vLink network to satisfy the required source, destination, bandwidth and data transfer limit. Since such a path is composed of vLinks, an is termed as virtual path (vpath) in order to differentiate it from path composed of overlay links. The vpath search procedure is a routing procedure in the vLink network. However, it is much simpler to route in a vLink network than in an overlay network, owing to the fact that a vLink network has a simpler configuration. In the system, there is a table which records the mapping between each existing vLink and its underlying path configuration. Therefore, the composition of the path for a new vLink is simply the combination of the path configuration of

each composing vLink.

Vlink Creation

In Chapter 3, two types of vLink creation are defined. The partial-partition creation establishes a space-sharing relationship between two vLinks, while the full-partition pattern sets up a time-sharing relationship. The vLink creation is more than creating an instance of Vlink class, but also establishes relationships between vLinks.

Implementation Steps

The procedure of creating a new vLink involves the following steps:

1. Creates a new vLink with the required source, destination, bandwidth and data transfer limit
2. Records the composing vLinks of this new vLink
3. Changes corresponding attributes of composing vLinks including the bandwidth, data transfer limit, and availability state
4. Registers those composing vLinks which have time-sharing relationship with the new vilnk in the taken-up vLink table of the CyberOrg Manager in order to guarantee that these vLinks are not available until the new vLink is consumed

5.1.2 Resource Allocation Task Distribution

The goal of the resource allocation task distribution procedure is to establish a network connection corresponding to a vLink. In this procedure, each node along the path which composes a connection is informed about their allocation tasks[1].

Implementation Steps

At a source node, implementation tasks involves:

1. Creates a distribution message which contains information of the entire path configuration of the to be established connection, and bandwidth allocation details (0 msg)
2. Keeps the data transfer limit of the corresponding vLink and sets up allocation tasks to be executed by the Transport Sender on this node (0 msg)
3. Sends the message asynchronously to the next node in the path of the connection (1 msg)
4. Waits for the finish message from the destination node (0 msg)

[1] In order to assist communication overhead analysis for each operation, the number of messages caused by each implementation step is given at the end of each step description in a bracket.

At an intermediate node, implementation tasks involves:

1. Sets up local allocation tasks to be executed by the local Transport Sender (0 msg)
2. Sends the message to the next node (1 msg)

At a destination node, implementation tasks involves:

1. Sets up local allocation tasks to be executed by the local Transport Sender (0 msg)
2. Sends a finish message to the source node of this path (1 msg)

Communication Overhead

Suppose there is a connection with n nodes, According to the description in the implementation steps there is one message triggered at each node. Therefore the total number of messages communicated in this procedure is n.

5.1.3 Post-Allocation Maintenance

The post-allocation maintenance procedure involves vLink consumption at the CyberOrgs level and path disestablishment at the network level.

vLink Consumption

The vLink consumption procedure is described in a recursive algorithm.

Algorithm 5 Consumed(Vlink vl_i, ActorName $facilitator_i$)

```
1:  if vl_i.getDataTransLimit() ≠ 0 then
2:      vl_i.setAvailState(FREE);
3:      sendMessage (facilitator_i, vLinkReturn);
4:  else
5:      removeFromSystem(vl_i);
6:  end if
7:  while vl_i.hasNextCreatorVLink() do
8:      Cvl_j = vl_i.nextCreaterVLink();
9:      if vl_i.getCreationPattern(Cvl_j) = FULL_PARTITION then
10:         CyberOrgManager.takenupVlinkTable.remove(Cvl_j);
11:         facilitator_j = CyberOrgManager.getFacilitator(Cvl_j);
12:         Consumed(Cvl_j, facilitator_j);
13:     end if
14: end while
```

In Algorithm 5, vl_i is the virtual link of which the consumption situation is under examination, and $facilitator_i$ is the facilitator of the cyberorg by which vl_i is held. The method of consumed is invoked when a vLink is consumed, the data transfer limit field of the consumed vLink is set to 0 before applying consumed() on it. Line 1 checks the current data transfer limit of vl_i. If the limit is zero, vl_i is consumed and removed from system record as in line 5. From line 7, the creator vLink of vl_i is being checking in order to change corresponding states. Due to vl_i can be created from multiple vLinks, a while loop is used. For Cvl_j , if vl_i is created from it in a full partition pattern which means all bandwidth of Cvl_j is given to vl_i for transferring vl_i. getDataTansLimit() amount of data. Before the consumption of vl_i, Cvl_j 's availability state is taken-up. When the consumed method is applied on Cvl_j, line 2 to 3 changes the availability state of Cvl_j from taken up to free and this vLink is returned back to the cyberorg which it belongs to by sending a 'vLinkReturn' message to the corresponding facilitator.

Connection Disestablishment

The connection disestablishment procedure is similar to the connection establishment procedure except the message passed is to remove the resource allocation information stored at each node along the path.

Implementation Steps

The source, intermediate and destination nodes are differentiated in this procedure as well.

At a source node, implementation tasks involves:

1. Creates a distribution message with the information of connection to be disestablished (0 msg)
2. Sends the message to the next node (1 msg)
3. Deletes local resource allocation task from the local Transport Sender (0 msg)

At an intermediate node, implementation tasks involves:

1. Deletes local resource allocation task from the local Transport Sender (0 msg)
2. Sends the message to next node (1 msg)

At a destination node, implementation tasks involves:

1. Deletes local resource allocation task from the local Transport Sender (0 msg)

Communication Overhead

There are $n-1$ messages used in this procedure if the number of nodes involved is n.

5.1.4 Resource Allocation In Different Scenarios

In the system, the resource allocation mechanism is activated in two scenarios. First, within a cyberorg, resource allocation happens before the start of a communication. Second, when a new cyberorg is to be isolated, the resource allocation procedure is carried out in order to enforce the transfer of requested vLinks from the original cyberorg to the new cyberorg.

Resource Allocation for Individual Communication

In the individual communication case, the resource allocation procedure is activated automatically, because it is difficult to predict when a communication starts. When an actor sends a message to a remote actor, the system checks whether the resource is allocated to this communication. If no resource is allocated, resource allocation procedure is invoked; otherwise the message is put into the corresponding message queue in the MessageManager.

Resource Allocation for Cyberorg Creation The creation of a new cyberorg may involve more than one vLink request. The resource allocation procedure is applied on each vLink request. However, it is not necessary to establish the network connection for each created vLink in implementation, because these vLinks have not been requested by any communication yet. In addition, the resource allocation procedures for creating the system cyberorg and system communication cyberorg are different from other cyberorg creation. Details are given in Section 5.2.2 and 5.3.1.

5.2 Actor Creation

According to the actor creation APIs introduced in Chapter 4, there are two types of actor creation: facilitator and application actor creation. In this section, implementation details of facilitator and application actor creation are described respectively.

5.2.1 Application Actor Creation

In CyberOrgs, resource concern is separated from functional concern of a computation. Therefore, an application actor is not aware the underlying cyberorg structure. It is the system that is aware of the cyberorg-related information related to each application actor, and such information is registered during the actor creation procedure. In this part, the steps involved in this procedure are given, and local and remote actor creation scenarios are differentiated.

Implementation Steps

In the local creation scenario, implementation steps includes:

1. A synchronous message is passed to the local facilitator[2]of the host cyberorg[3] by which the actor to be created is held, to add the name of the new actor to its hosted actor list. Synchronous message passing is used here because they system needs to make sure every application actor belongs to a specific cyberorg so that resource requests initiated by the new actor can be processed properly. A new actor is hosted by the cyberorg which its creator, either an application actor or a facilitator belongs to (2 msg)

2. An instance of this new actor is created and started. (0 msg)

In the remote scenario, implementation steps includes:

1. If the master facilitator of the host cyberorg which this new actor belongs is not local, two synchronous messages are passed to both the local worker and remote master facilitator of the host cyberorg to add the name of the new actor to their hosted actor list (4 msg)

2. Else if the master facilitator of the host cyberorg is local, one synchronous message is sent to it to add the name of the new actor to its hosted actor list (2 msg)

3. An instance of this new actor is created and started. (0 msg)

Communication Overhead

The number of messages triggered by local actor creation is 2, because asynchronous communication requires a reply message. The remote creation causes 2 messages when the local worker facilitator is same to the master facilitator, and 4 messages otherwise.

5.2.2 Facilitator Actor Creation

Due to the special function of a facilitator actor to the cyberorg it represents, besides common actor creation requirements, it also carries out cyberorg-related tasks, such as cyberorg registration and cyberorg contents set up.

Implementation Steps

This procedure involves the following steps:

1. Adds the cyberorg, which the facilitator under discussion represents, to local cyberorg hierarchy (0 msg)

2. Asynchronously informs the local facilitator of the parent cyberorg (if there is any[4]) to add this new cyberorg as a child. Asynchronous communication is used here since late child list

[2]Local facilitator means local worker facilitator.
[3]The cyberorg by which an actor is held is called the host cyberorg to this actor
[4]The system cyberorg does not have a parent cyberorg, because the root cyberorg is a conceptual cyberorg and is not implemented in the system

update will not influence any other operation requests received by the parent cyberorg (1 msg)

3. If the facilitator to be created is a master facilitator, sets up global[5] contents including eCash account, resource (vLinks) and actors according to the contract under which this new cyberorg is created (0 msg)

4. Else if the facilitator to be created is a worker facilitator, sets up local eCash account and actors if any[6] (0 msg)

5. Sets or changes[7] the record of the previous host cyberorg, which these isolated actors belong to before the isolation, to the new cyberorg (0 msg)

6. Creates an instance of the new facilitator and starts it (0 msg)

Communication Overhead

The total number of messages triggered by a facilitator actor creation operation is 1, because asynchronous communication does not need a reply.

5.3 CyberOrg Creation

In Chapter 4, the API of cyberorg creation is given. However, the difference between the creation of the system and other cyberorgs is not fully explained. This part describes system cyberorg and isolated cyberorg creation separately.

5.3.1 System Cyberorg Creation

Generally, the creation of a cyberorg involves facilitator creations and contents set up. Due to the distributed nature of a cyberorg in the prototype system, the distribution scope has to be determined. As the system cyberorg holds entire system resource, the distribution of this cyberorg covers all nodes in the system. The number of all nodes in the system is represented by N. In addition, the system creation is part of system initialization.

Implementation Steps

This procedure involves the following steps:

1. First, a master facilitator of the system cyberorg is created on the node where the creation request is received. This is a facilitator creation procedure. ($N(facilitatorCreation) = 1$ msg)

[5] By 'global', it is meant global to the new cyberorg
[6] It is possible a cyberorg is created with zero actors
[7] In case of isolation.

2. Then, the master facilitator broadcasts worker facilitator creation messages to all the other system nodes, and synchronously waits until all worker facilitators are created. Synchronization is necessary here for two reasons. First of all, the initialization phase does not proceed until the completion of the system cyberorg creation. Secondly, when a cyberorg is under creation, its state is set to unknown in order to avoid the inconsistency caused by processing any service request received. It is at the end of a creation procedure, the state of a cyberorg is changed to be active ($(N-1)(2+N(facilitatorCreation)) = 3(N-1)$ msg)

Communication Overhead

There are $3N-2$ messages caused by a system cyberorg creation operation.

5.3.2 Isolated Cyberorg Creation

An isolated cyberorg covers a few nodes in the system, according to the network resource receiving from its parent cyberorg. Therefore the distribution scope of an isolated cyberorg is smaller than the system cyberorg. The number of nodes covered by an isolated cyberorg is denoted n.

Implementation Steps

This procedure includes the following steps:

1. First, a master facilitator of the isolated cyberorg is created on the node where the isolation request is received ($N(facilitatorCreation) = 1$)

2. Then, the master facilitator broadcasts worker facilitator creation messages to all the other nodes covered by the isolated cyberorg, and synchronously waits until all worker facilitators are created. The reason for synchronization is similar to the second reason of system cyberorg creation ($(n-1)(2+N(facilitatorCreation)) = 3(n-1)$ msg)

Communication Analysis

The number of message passing because of isolated cyberorg creation is $3n-2$.

5.4 Distributed Isolation

Cyberorg isolation is different from cyberorg creation. Although, the creation of a cyberorg is an important step in an isolate operation, isolation does more than only creating a new cyberorg. An isolate operation changes the structure of the isolating cyberorg as well. In addition, the distributed nature of a cyberorg poses challenges to implementation. How to coordinate facilitators of the isolating cyberorg to carry out isolation? Which facilitators are involved? In addition, the

facilitators of a cyberorg are also distributed. How should they communicate? These problems are raised up and have to be properly handled in the implementation.

According to APIs developed in Chapter 4, implementation details of system communication cyberorg isolation and application cyberorg isolation are presented in this part. As resource allocation calculation and scheduling changing are required for an isolate operation, corresponding computation overhead analysis is given in addition to communication overhead analysis.

5.4.1 System Communication Cyberorg Isolation

Isolation happens after the generation of a contract. In the case of system communication cyberorg isolation, the system cyberorg is the parent cyberorg which generates the contract and performs the isolation. Because a system communication cyberorg holds all the resource reserved for system communications, this cyberorg covers all the nodes in the system. Therefore, every facilitator representing the system cyberorg on each node is involved in this isolation, and these facilitators coordinate in a master-worker way. Specifically, there are two roles involved in this operation: the master facilitator and worker facilitators of the system cyberorg. The master facilitator carries out 'isolateSysComCyberOrg' task, while the worker facilitator performs 'workerIsolateSysComCyberOrg' task. Furthermore, a certain percentage of the bandwidth of each link in the system is reserved for system communication. The procedure of system communication cyberorg isolation is the second and the last step of system initialization. It is after this step, the system is started and the underlying communication is switched from TCP to UDP.

Implementation Steps

In the 'isolateSysComCyberOrg' task, the master facilitator of the system cyberorg performs the following tasks:

1. Updates its record of resource and eCash account by deducting the correspondent isolated amount according to the contract generated. (0 msg)

2. Performs the 'reserveResource' task which reserves resource for system communication. On the node where this master facilitator resides, local CyberOrg Manager computes the shortest paths[8] from this node to any other reachable node by using Breadth-First-Search algorithm [43], and the next hop address for each shortest path is kept in this node as well. (0 msg)

3. Creates the system communication cyberorg[9] ($N(cyberorgCreation)) = 3N - 2$ msg)

4. Synchronously broadcasts to all worker facilitators in the system cyberorg to do 'workIsolateSysComCyberOrg' task. The reason for synchronous communication is because the mas-

[8] with smallest hop counts
[9] Please refers to cyberorg creations

ter facilitator has to make sure every worker facilitator has finished their task and starts the local system on each node ($(N-1)(2+N(workIsolateSysComCyberOrg)) = 2(N-1)$ msg)

5. After receiving replies from all worker facilitators, starts the system. (0 msg)

$N(isolateSysComCyberOrg) = 5N - 4$

In the 'workerSysComCyberOrg' task, a worker facilitator of the system cyberorg performs the following tasks:

1. Deducts local eCash account (0 msg)
2. Reserves system resource (this is similar to step 2 in isolateSysComCyberOrg) (0 msg)
3. Starts local system and switches the underlying protocol from TCP to UDP (0 msg)
4. Returns a reply message to the master facilitator to indicate the completion of this task (already counted in isolateSysComCyberOrg)

$N(workerSysComCyberOrg) = 0$

Communication Overhead

The number of message passing because of system communication cyberorg isolation is $5N - 4$.

Computation Overhead

The computation overhead is mainly caused by resource reservation. In this step, BFS is used to calculate the path from the source node to every other reachable node in the system. The cost of running BFS on a graph G(V, E) is $O(V + E)$ [43], V is the number of nodes, and E is the number of edges. There are N nodes and E links in the system, so the cost of reserving resource at each node is $O(N + E)$.

5.4.2 Application Cyberorg Isolation

A new application cyberorg is isolated from an existing cyberorg based on an isolation contract. In the isolation contract, vLinks, actors and eCash to be given to the new cyberorgs are specified. In an application cyberorg isolation, facilitators on the nodes where isolated actors exist are involved. Similar to the system communication cyberorg, they coordinate in a master-worker pattern. If the isolate request is originated from the node where the master facilitator resides, the 'masterIsolate' procedure is performed by the master facilitator and 'remoteWorkerIsolate' is done by worker facilitators. Otherwise, the 'workerIsolate' task is taken by the worker facilitator on the node where the isolate request is issued, the 'remoteMasterIsolate' is carried out by the master facilitator and 'remoteWorkerIsolate' task is performed by the other involved worker facilitators.

Implementation Steps

In the 'masterIsolate' task, the master facilitator of the isolating cyberorg performs the following tasks:

1. Transfers resources allocated to the isolated actor communications to the isolated cyberorg (0 msg)

2. Calculates resource allocation for new vLink requests and transfers these resources to the new cyberorg.

3. Transfers eCash to the isolated cyberorg. (0 msg)

4. Removes isolated actors from both the global hosted actor list and local hosted actor list (0 msg)

5. Creates a new cyberorg. $(N(cyberorgCreation)) = 3n - 2$ msg)

6. Asynchronously informs relevant worker facilitators to perform 'remoteWorkerIsolate' task. Asynchronous communication is used to avoid dead lock. It is possible that another worker-initiated isolation happens at the same time $((n-1)(1 + N(remoteWorkerIslate)) = n - 1$ msg)

$N(masterIsolate) = 4n - 3$

In the 'workerIsolate' task, the worker facilitator of the isolating cyberorg performs the following tasks:

1. Synchronously informs the remote master facilitator to isolate by sending a remoteMaster-Isolate message, and waits for the remote master facilitator to decide whether this isolate request can be accepted $(2 + N(remoteMasterIsolate) = n + 1$ msg)

2. If a deny message is received, cancels this request. The message indicates the isolate request can not be satisfied right now (0 msg)

3. Else if an ok message is received, transfers local eCash to the isolated cyberorg (0 msg)

4. Creates a new cyberorg according to the returned isolation contract $(N(cyberorgCreation) = 3n - 2$ msg)

5. Returns the new created cyberorg (0 msg)

$N(workerIsolate) = 4n - 1$

In the 'remoteMasterIsolate' task, the master facilitator performs the following tasks:

1. Checks if the isolate request can be satisfied right now. An import criterion is the consistency of system information, such as whether the requested actors to be isolated are currently belonging to the cyberorg (0 msg)

2. If yes, generates an isolation contract (0 msg)

3. Transfers resources allocated to the isolated actor communications to the isolated cyberorg (0 msg)

4. Calculates resource allocation for new vLink requests and transfers these resources to the new cyberorg.

5. Transfers eCash to the isolated cyberorg. (0 msg)

6. Removes isolated actors from the global hosted actor list and local hosted actor list (0 msg)

7. Asynchronously informs relevant worker facilitators to perform 'remoteWorkerIsolate' task. Asynchronous communication is used to avoid dead lock. It is possible that another worker-initiated isolation happens at the same time $((n-1)(1+N(remoteWorkerIsolate)) = n-1)$

8. returns the reply message which contains the generated isolation contract (already counted in 'workerIsolate')

$N(remoteMasterIsolate) = n-1$

In the 'remoteWorkerIsolate' task, the worker facilitator performs the following tasks:

1. Deducts local eCash (0 msg)

2. Removes isolated actors from local hosted actor list (0 msg)

$N(remoteWorkerIsolate) = 0$

Communication Overhead

The number of message passing because of master-initiated isolation $4n-3$, and $4n-1$ for worker-initiated isolation.

Computation Overhead

The computation overhead is mainly caused by calculating resource allocation for request specified in an isolation contract. If l is the number of new vLinks, c_l is the cost for resource allocation calculation for an individual vLink request. In the system, resource allocation is calculated by applying the Dijkstra's algorithm. The cost of Dijkstra's algorithm is $O(V^2)$ (if the direct network graph is G(V, E), V is the number of vertices and E is the number of edges)[43]. Suppose a cyberorg covers n nodes, c_l is $O(n^2)$, and the total cost for computing l requests is $O(l(n^2))$. In the case

that a vLink is created from existing vLinks, the computation cost is much cheaper, because the involved number of node is much less than n.

5.5 Distributed Assimilation

The assimilate operation involves an assimilating cyberorg and its parent cyberorg. The assimilating cyberorg initiates assimilate request, and its parent cyberorg decides whether to accept or decline the assimilate request. Here behaviors of an assimilating cyberorg and its parent cyberorg are described, as well as how they cooperate to accomplish a distributed assimilate operation.

In an assimilate operation, all facilitators of the assimilating cyberorg are involved and only those parent facilitators which coexist with these assimilating facilitators are involved as well, facilitator coordinate in a master-worker style. In the current implementation, an assimilate operation involves four different roles which are the worker and master facilitators of an assimilating cyberorg, and the worker and master facilitators of the parent cyberorg. Correspondingly, the worker and master assimilating facilitators carry out 'master Assimilation' and 'workerAssimilation' tasks, while the parent master and worker facilitators carry out 'acceptMasterAssimilation' and 'acceptWorkerAssimilation' tasks.

Implementation Steps

When a cyberorg assimilate request is issued from the node where the master facilitator of this cyberorg exists. The 'masterAssimilate' task is carried out by the master facilitator.

In the 'masterAssimilate' task, a master facilitator of the assimilating cyberorg performs the following tasks:

1. Suspends all local actors (0 msg[10])

2. Finds the local parent facilitator and parent master facilitator. If they are the same facilitator, an asynchronous message to invoke the 'acceptMasterAssimilate' task is sent to the local parent facilitator. Otherwise, two asynchronous messages, one to invoke the 'acceptWorkerAssimilate' task is sent to the local parent facilitator and the other to invoke 'acceptMasterAssimilate' task is sent to the parent master facilitator ($2+N(acceptMasterAssimilate)+N(acceptLocalAssimilate) = 2$ msg)

3. Synchronously informs all worker facilitators of the assimilating cyberorg to do 'remoteWorkerAssimilate' task by sending broadcast messages ($((n-1)(2+N(remoteWorkerAssimilate))) = 3(n-1)$ msg)

[10]Because the local Actor Manager has a record of each actor residing on the node, suspend is implemented as a local method invocation

4. Deregisters the assimilated cyberorg from the local cyberorg hierarchy tree (0 msg)

5. After receiving replies from all worker facilitators, destroys itself (0 msg)

$N(masterAssimilate) = 3n - 1$

The 'remoteWorkerAssimilate' task is invoked after receiving corresponding message sent by the master facilitator.

In the 'remoteWorkerAssimilate' procedure, the worker facilitator of the assimilating cyberorg performs the following tasks:

1. Suspends all local actors (0 msg)

2. Finds local parent facilitator and asynchronously sends acceptLocalAssimilate message to it $(1 + N(acceptLocalAssimilate) = 1)$

3. Deregisters the assimilated cyberorg from the local cyberorg hierarchy tree (0 msg)

4. Destroys itself (0 msg)

$N(remoteWorkerAssimilate) = 1$

When a cyberorg assimilate request is issued from the node where a worker facilitator of this-cyberorgg exists. The 'workerAssimilate' task is carried out by the worker facilitator.

In the 'workerAssimilate' task, a worker facilitator of the assimilating cyberorg performs the following tasks:

1. Synchronously informs remote master facilitator to carry out 'remoteMasterAssimilate' task. This synchronous communication is necessary because it is possible that the cyberorg under discussion is not ready for assimilating. For example, an isolate operation of the cyberorg has not finished yet. It is only the master facilitator be able to make a decision $(2 + N(remoteMasterAssimilate) = 3n - 2$ msg)

2. If an deny message is returned, cancels this assimilate request and return (1 msg)

3. Else if an OK message is received, suspends all local actors (1 msg)

4. If the local parent facilitator is not a master facilitator, informs it to perform 'acceptLocalAssimilationg' task by sending asynchronous message. $(1 + N(acceptLocalAssimilationg) = 1$ msg)

5. Deregisters the assimilated cyberorg from the local hierarchy tree (0 msg)

6. Destroys itself (0 msg)

$N(workerAssimilate) = 3n$

The 'remoteMasterAssimilate' task is invoked in the worker-initiated assimilation scenario by receiving the remoteMasterAssimilate message sent from a remote worker facilitator (call it informer).

In the 'remoteMasterAssimilate' task , the master facilitator of the assimilating cyberorg performs the following tasks:

1. Checks whether the cyberorg is ready for assimilation (0 msg)
2. If not, sends a deny message and returns (counted in workerIsolate)
3. Else if yes, suspends all local actors (0 msg)
4. Finds the local parent facilitator and parent master facilitator. If they are the same facilitator, an asynchronous message to invoke 'acceptMasterAssimilate' task is sent to the local parent facilitator. Otherwise, two messages, one for 'acceptLocalAssimilate' task and the other one for 'acceptMasterAssimilate' task are sent to the parent local and master facilitators respectively. $(2 + N(acceptMasterAssimilate) + N(acceptLocalAssimilate) = 2$ msg)
5. Synchronously informs all worker facilitators of the assimilating cyberorg (except the informer) to carry out 'remoteWorkerAssimilate' task by broadcasting $((n-2)(2+N(remoteWorkerAssimilate))$ $3(n-2)$ msg)
6. Deregisters the assimilated cyberorg from the local cyberorg hierarchy tree (0 msg)
7. Destroys itself (0 msg)
8. An OK message is returned to the informer to indicate the finishing of this distributed assimilate operation. (already counted in 'workerIsolate')

$N(remoteMasterAssimilate) = 3n - 4$

The 'acceptMasterAssimilate' task is performed by the master facilitator of the parent cyberorg of an assimilating cyberorg.

In the 'acceptMasterAssimilate' task, the master facilitator of the parent cyberorg performs the following tasks:

1. Adds the assimilated eCash to its eCash account (0 msg)
2. Adds the assimilated resource (0 msg)
3. Adds the assimilated actors to both the global and local hosted actor list (0 msg)
4. Changes the record of the parent cyberorg of the assimilating cyberorg to be this cyberorg (0 msg)

5. Adds the assimilated cyberorg from its hosted cyberorg list (0 msg)

$N(acceptMasterAssimilate) = 0$

The 'acceptLocalAssimilate' task is performed by the worker facilitator of the parent cyberorg of an assimilating child cyberorg.

In the 'acceptLocalAssimilate' task, the worker facilitator of the parent cyberorg performs the following tasks:

1. Adds the assimilated local actors to its local hosted actor list
2. Changes the record of the parent cyberorg of the assimilating cyberorg to be the cyberorg which this worker facilitator represents (0 msg)
3. Adds the assimilated cyberorg to its hosted cyberorg list (0 msg)

$N(acceptLocalAssimilate) = 0$

Communication Overhead

The number of messages caused by master-initiated assimilation is $3n - 1$, and $3n$ for worker-initiated assimilation.

Computation Overhead

In an assimilate operation, vLinks held by the assimilated cyberorg are returned to its parent cyberorg. Because these vLinks are already computed, it is reasonable to keep them by adding them to the parent cyberorg's vLink table.

5.6 Distributed Migration

In a migrate operation, involved cyberorgs have a role of customer or supplier. The cyberorg which requests migrate is a customer cyberorg, and the cyberorg which allows the migration is a supplier cyberorg. In addition, there are two phases in a migration. Suppose there is a yellow page service, and the supplier cyberorg is known already. The first phase is negotiation. It is only after having a successful negotiation that the customer cyberorg is allowed to migrate to the supplier cyberorg. In this part these two phases are described separately.

5.6.1 Negotiation Phase

In the negotiation phase, a customer cyberorg facilitator (no matter worker or master) carries out 'initiateNegotiation' task, and the master facilitator in the supplier cyberorg carries out 'process-Negotiation' task. No new computations are carried out in an assimilate operation.

Implementation Steps

In the 'initiateNegotiation' task, the facilitator asynchronously informs the master facilitator in the supplier cyberorg to carry out 'processNegotiation' task. There is no need to have synchronous communication because it is the supplier cyberorg who decides whether to accept this migration and when to start it.

In the 'processNeogitation' task, the master facilitator of the supplier cyberorg performs the following tasks:

1. Calculates resource allocation for the vLink request specified in the proposed migration contract (0 msg)
2. Decides whether to accept this request based on its negotiation policy. The implementation supports simple decision making strategy which accepts the request as long as this cyberorg can satisfy proposed resource requests and the proposed price is not lower than the set price (0 msg)
3. If the request is accepted, resource is reserved for this migration. l is used to denote the number of vLink requests, and $vn(i)$ represents the number of node involved in a vLink i ($N(resourceAllocation) = \sum_{i=1}^{l} vn(i)$ msg)
4. Sets price for the resource to be transferred in the contract (0 msg)
5. Asynchronously informs the facilitator of customer cyberorg to do 'postNegotiation' task. Again there is no need to be synchronous because the negotiation procedure is finished.($N(postNegotiation)$ msg)
6. If the request is denied, sets the accepted price and comments in return message and asynchronously informs 'postNegotiation' task ($N(postNegotiation)$ msg)

$N(processNegotiation) = 5n + l(vn + 1)$

In the 'postNegotiation' task, the facilitator of the customer cyberorg checks if the negotiation is accepted by examining the comments in the return message and performs the following tasks:

1. If accepted, invokes migration procedure ($N(Migrate)$)
2. Else if not, prints out the comment (0 msg)

$N(postNegotiation) = 5n$

5.6.2 Migration Phase

In the migration phase, five roles are involved which are the master and worker facilitators in the migrating/customer cyberorg, facilitators (no matter worker or master) in the parent cyberorg of

this migrating cyberorg, and master and worker facilitators in the supplier cyberorg. The master facilitator in the migrating cyberorg performs 'masterMigrate' or 'remoteMasterMigrate' tasks which depends on where the migrate request comes from, and the worker facilitator does "workerMigration' or 'remoteWorkerMigrate' tasks. The facilitators in the parent cyberorg carry out 'acceptMigration', and the master and worker facilitators in the supplier cyberorg performs 'allowMasterMigration' and 'allowLocalMigration' tasks respectively.

Implementation Steps

When a migrate request is issued from the node where the master facilitator of the customer cyberorg exists, the 'masterMigrate' task is performed by this master facilitator.

In the 'masterMigrate' task, the master facilitator of the migrating/customer cyberorg performs the following tasks:

1. Changes cyberorg state from active to migrating. This makes the migrating cyberorg to deny other primitive operation requests during the migration procedure (0 msg)
2. Retrieves local parent facilitator and asynchronously informs it to do the 'acceptMigration' task ($1 + N(acceptMigration) = 1$ msg)
3. Asynchronously informs remote worker facilitators to do 'remoteWorkerMigrate' ($(n-1)(1 + N(remoteWorkerMigrate)) = 2(n-1)$ msg)
4. Synchronously informs the master facilitators in the supplier cyberorg to do 'allowMasterMigration' ($2 + N(allowMasterMigration) = 3n - 1$ msg)
5. Adds transferred resource to the resource table in this migrating cyberorg (0 msg)
6. Changes the local cyberorg hierarchy tree (0 msg)

$N(masterMigrate) = 5n - 2$

When the migrate request is issued from the node where a worker facilitator of the customer cyberorg resides, the 'workerMigrate' is performed by the worker facilitator.

In the 'workerMigrate' task, the worker facilitator of the migrating/customer cyberorg performs the following tasks:

1. Changes cyberorg state from active to migrating(0 msg)
2. Asynchronously informs remote master facilitator to do 'remoteMasterMigrate' ($1+N(remoteMasterMigrate$ $5n - 1$ msg)
3. Asynchronously informs local parent facilitator to do 'acceptMigrate' task ($1+N(acceptMigrate) =$ 1 msg)

$N(workerMigrate) = 5n$

In the 'remoteMasterMigrate' task, the master facilitator of the migrating/customer cyberorg performs the following tasks:

1. Sets the state of this migrating cyberorg from active to migrating (0 msg)
2. Retrieves local parent facilitator and asynchronously informs local parent facilitator to 'accept migration' ($1 + N(acceptMigration) = 1$ msg)
3. Pays for the resource and deducts the eCash account (0 msg)
4. Synchronously informs the master facilitator in the supplier cyberorg to do 'allowMasterMigration' ($N(allowMasterMigration) + 2 = 3n - 1$ msg)
5. Adds transferred resource to the resource record of this cyberorg (0 msg)
6. Asynchronously informs remote worker facilitators to do 'remoteWorkerMigrate' ($(n-1)(1 + N(remoteWorkerMigrate)) = 2(n-1)$ msg)
7. Changes its local hierarchy tree (0 msg)
8. If the master facilitator of the server cyberorg is local, changes the parent cyberorg of this migrating cyberorg at local Cyberorg Manager (0 msg)

$N(remoteMasterMigrate) = 5n - 2$

In the 'remoteWorkerMigrate' task, the worker facilitator of the migrating cyberorg performs the following tasks:

1. Sets the state of this migrating cyberorg from active to migrating (0 msg)
2. Retrieves local parent facilitator and asynchronously informs local parent facilitator to do 'acceptMigration' ($1 + N(acceptMigration) = 1$ msg)

$N(remoteWorkerMigrate) = 1$

The 'acceptMigration' simply removes the migrating cyberorg from child list. ($N(acceptMigration) =$ 0 msg)

In the 'allowMasterMigration', the master facilitator of the supplier cyberorg performs the following tasks:

1. Receives payment and increments its eCash account (0 msg)
2. Asynchronously informs relevant worker facilitators to do 'allowWorkerMigration' ($(n-1)(1+ N(allowWorkerMigration)) = 3(n-1)$ msg)
3. Adds the migrating cyberorg to hosted cyberorg list (0 msg)

$N(allowMasterMigration) = 3(n-1)$

In the 'allowWorkerMigration', the worker facilitator of the supplier cyberorg performs the following tasks:

1. Adds the migrating cyberorg to child list (0 msg)
2. Asynchronously informs the local worker facilitator of the migrating cyberorg to do 'postWorkerMigration' ($1 + N(postWorkerMigration) = 2$ msg)

$N(allowWorkerMigration) = 2$

In the 'postWorkerMigration' task, the worker facilitator of the migrating cyberorg performs the following tasks:

1. Changes its local hierarchy (0 msg)
2. Asynchronously informs the master facilitator to do 'postMasterMigration' ($1+N(postMasterMigration) = 1$ msg)

$N(postWorkerMigration) = 1$

In the 'postMasterMigration', the master facilitator of the migrating cyberorg performs the following tasks:

1. Accumulates the reply number until the all replies are received (0 msg)
2. Updates the states of this migrating cyberorg from migrating to active (0 msg)

$N(postMasterMigration) = 0$

Communication Overhead

The numbers of messages passing because of master-initiated migration or worker-initiated migration are $5n - 2$ and $5n$ respectively.

Computation Overhead

The computation overhead of migration results from resource allocation calculation due to vLinks requested in the migration contract. Similarly to Isolation, the cost is $O(l(n^2))$. n is the number of node covered by the supplier cyberorg.

5.7 Conclusion

System overhead is caused by invocation of CyberOrgs primitives. Because cyberorgs in the prototype system is internally distributed, primitive operations are implemented in a distributed manner

Table 5.1: System Overhead (n is the number of nodes covered by a cyberorg; l is the number of vLink requests)

Operation	Communication Overhead	Communication Overhead	Computation Overhead
	Local Messages	Remote Messages	
Isolate	$O(n)$	$O(n)$	$O(ln^2)$
Assimilate	$O(n)$	$O(n)$	
Migrate	$O(n)$	$O(n)$	$O(ln^2)$
Allocate	0	$O(l)$	$O(n^2)$

as well. Correspondingly, the invocation of a primitive operation causes system message communication and may also result in resource allocation computation. Table 5.1 summarizes communication and computation overhead for each primitive operation.

For communication overhead, the number of triggered messages is counted to demonstrate the amount of system communication caused by a primitive operation. Table 5.1 shows that the amount of system communication for a primitive operation is linear with the number of nodes covered by a cyberorg. In addition to the amount of system data transferring, system resource consumed by these data transferring is also considered. Here, local messages are differentiated from remote messages because it is the latter that consume system resource. A local message is directly delivered to a corresponding thread. Since remote messages may have different source and destination requests, vLinks consumed by transferring these messages are different. However, as system resources are reserved in the system communication cyberorg in advance, these messages do not compete with application messages for resources.

Besides communication overhead, a primitive operation may also require recalculating resource allocation. Both the isolate and migrate operations has to calculate the resource allocation in order to create the new cyberorg. Suppose there are l resource requests, the overhead for calculating resource allocation for these requests is $O(ln^2)$. The number of requests is related to the number of actors in each cyberorg. The more actors, the more communications may be initiated between them. As a result, the computation overhead for isolate and migrate is related to the number of actors and nodes covered by a cyberorg.

Since an application cyberorg covers a few nodes in the system. System expansion does not necessarily influence the configuration of an existing cyberorg. In addition, the vLink abstraction enables creating new vLinks by composing existing vLinks and thus simplifies the network configuration of underlying a cyberorg and reduces overhead for resource allocation computation. Therefore, this implementation is scalable.

Chapter 6

Experiment Results

In the previous chapter, communication and computation overheads of CyberOrgs hierarchical control implementation are analyzed. As application level resource requests are eventually enforced in the network by the resource scheduling layer, a set of experiments is carried out to evaluate the performance of the scheduling scheme developed in this layer.

6.1 Experiment Design

As described in Section 4.3.2, the scheduling scheme designed in the resource scheduling layer is composed of three types of scheduling: message, packet and thread scheduling. This scheduling scheme is aimed to achieve fine-grained per-flow rate-based bandwidth allocation. Accordingly, in experiment, performance of bandwidth allocation is evaluated in the following scenarios: 1) a single flow has different bandwidth requests; 2) a single flow is scheduled under different time granularities; 3) multiple co-existing flows are under scheduling. These experiments are performed in two-node, three-node and multi-node environments.

6.1.1 Experiment Settings

The assumption of CyberOrgs-based network resource management model is a closed dedicated system environment, in which resources are predictable in the system. Correspondingly, these

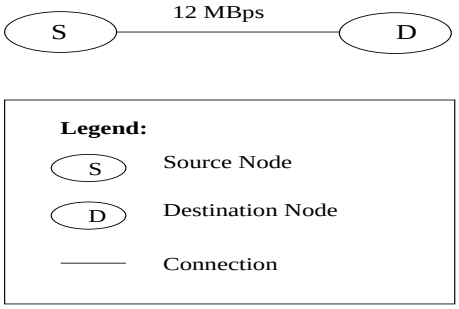

Figure 6.1: Two Node Topology

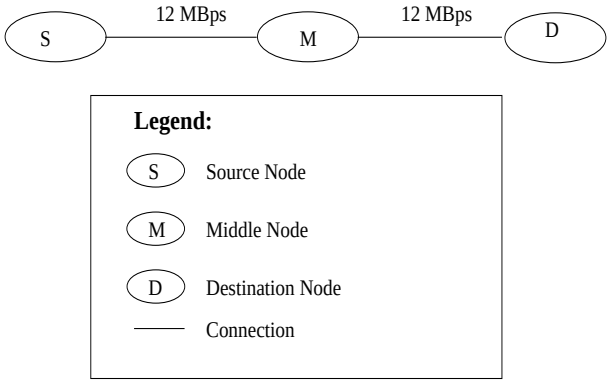

Figure 6.2: Three Node Topology

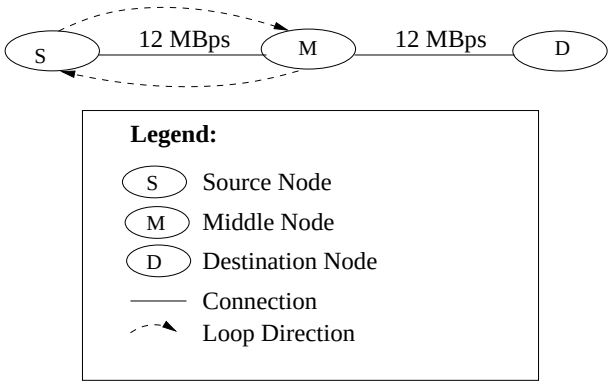

Figure 6.3: Multiple Node Simulation Topology

experiments are carried out in a closed system with a number of computers connected through a router.

The two-node system configuration is shown in Figure 6.1. This configuration is composed by connecting two computers through a router. The source node is an iMac G5 computer which has a 2.1GHz PowerPC G5 processor and 512MB memory. The destination node is a Mac mini machine with a 1.42GHz PowerPC G4 and 512MB memory. The router which is not shown in the figure is a 2.4 GHz Linksys router. Available bandwidth between S and D is approximately 12 MBps and this value is tested by using Pathload which is a tool for estimating the available bandwidth of an end-to-end path from a host S (sender) to a host R (receiver) [44].

As to the three-node environment, another iMac G5 computer is connected through to compose the configuration shown in Figure 6.2. The value of available bandwidth between S and M, M and D is both approximately 12 MBps. Figure 6.3 shows the topology used for establishing multi-node simulation. In this simulation, a 7 node environment is simulated by creating three loops between S and M (S, M, S, M, S, M), and D serves as the destination where data is collected.

6.2 Result Analysis

$$e = \frac{\sum_{i=1}^{n} \frac{|b_i - q|}{q}}{n}$$

In this section, experiment results are presented. Two metrics are used: bandwidth and inaccuracy, to analyze the performance of this scheduling scheme. The metric of bandwidth characterizes the actual bandwidth allocated to a flow, which is the average data transfer rate and is measured by accounting the packet receiving rate at the recipient‘s side. Bandwidth allocated to a single flow is measured within each 200 millisecond and the overall observation period is about 2 minutes. The inaccuracy quantifies the oscillation of bandwidth allocation, and the equation shown at the beginning of this section gives the formula. In the formula, e is the average inaccuracy. b_i represents actual allocated bandwidth during the ith 200 millisecond, q is the requested bandwidth of this flow, and n is the number of 200 milliseconds during the 2 minutes observation time period.

6.2.1 Two-Node Experiments

In the two-node environment, three experiments are performed: 1) to test the performance of bandwidth allocation to a single flow under a fixed time granularity with different bandwidth requests; 2) to evaluate the influence of different time granularity on scheduling performance; and 3) to examine the effect of co-existing flows on scheduling performance.

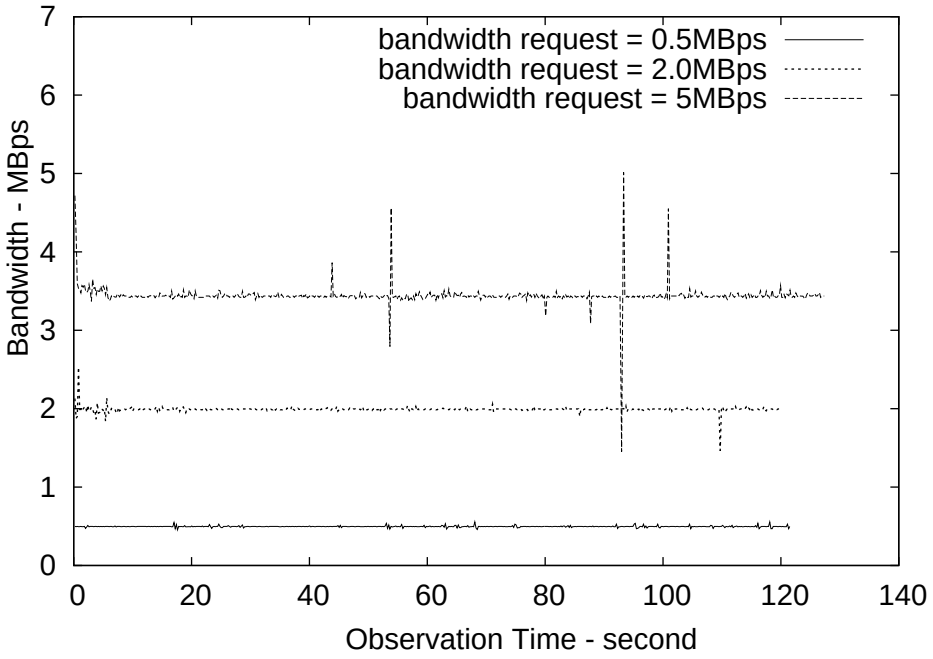

Figure 6.4: Performance Comparison of Single Flow Scheduling with Different Bandwidth Requests (bandwidth requests: 0.5MBps-5MBps; time granularity: 200 milliseconds; observation time: 2 minutes; environment: two-node)

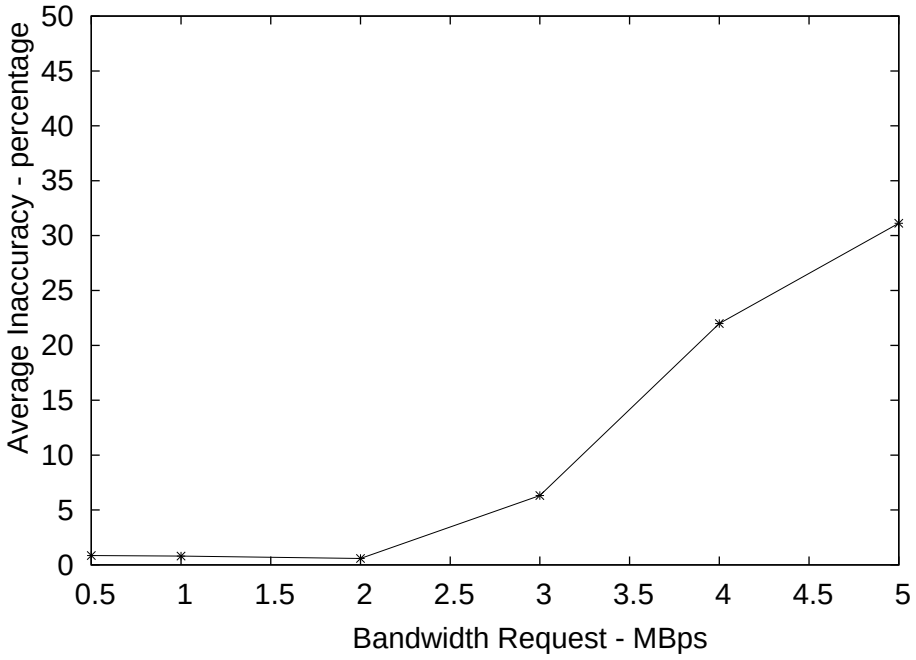

Figure 6.5: Inaccurray Comparison of Single Flow Scheduling with Different Bandwidth Requests

Table 6.1: Inaccuracy Comparison of Scheduling With Different Bandwidth Requests

Bandwidth Request	Inaccuracy
0.5 MBps	0.855
1 MBps	0.801
2 MBps	0.58
3 MBps	6.313
4 MBps	21.989
5 MBps	31.119

Single Flow Scheduling With Different Bandwidth Requests

In the first experiment, the time granularity of scheduling cycle is 200 milliseconds, and bandwidth request of the single flow under testing ranges from 0.5 MBps to 5 MBps.

Figure 6.4 shows the result of bandwidth allocation to a single flow with different bandwidth requests, and Figure 6.5 demonstrates average inaccuracy for each bandwidth request. As illustrated in Figure 6.5, when a bandwidth request is below 3 MBps, the actual allocated bandwidth to the flow is close to the requested value. The inaccuracy of bandwidth allocation in these cases is under 1%. By contrast, when bandwidth request exceeds 2.5 MBps, actual allocated bandwidth during observation period significantly deviates from the requested value.[1] This result reflects the inclination which is the higher the bandwidth request is, the more serious the actual allocation deviates from the requested value. Data for inaccuracy analysis is give in Table 6.1.

The phenomenon of serious deviation when bandwidth requests is greater than 2.5 MBps is caused by packet loss. Specifically, in each scheduling cycle, the number of packets to be scheduled is the product of requested bandwidth and scheduling time granularity. These packets are sent to the network as a bulk. When the bandwidth request is higher, the bulk size is larger which causes more packet loss.

Single Flow Scheduling With Different Time Granularities In this experiment, the time granularity of scheduling cycle is changed to analyze the influence of these changes on scheduling performance. 200, 400 and 800 milliseconds are chosen as different granularities. This experiment is repeated on a single flow for different bandwidth requests, which are 0.5 MBps, 1 MBps and 2 MBps.

The result is shown in Figure 6.6, 6.7 and 6.8. For the 0.5 MBps bandwidth request, the corresponding inaccuracy is reduced with the increment of time granularity. However, this trend does

[1] when the inaccuracy is over 10%, it is considered as significantly deviation

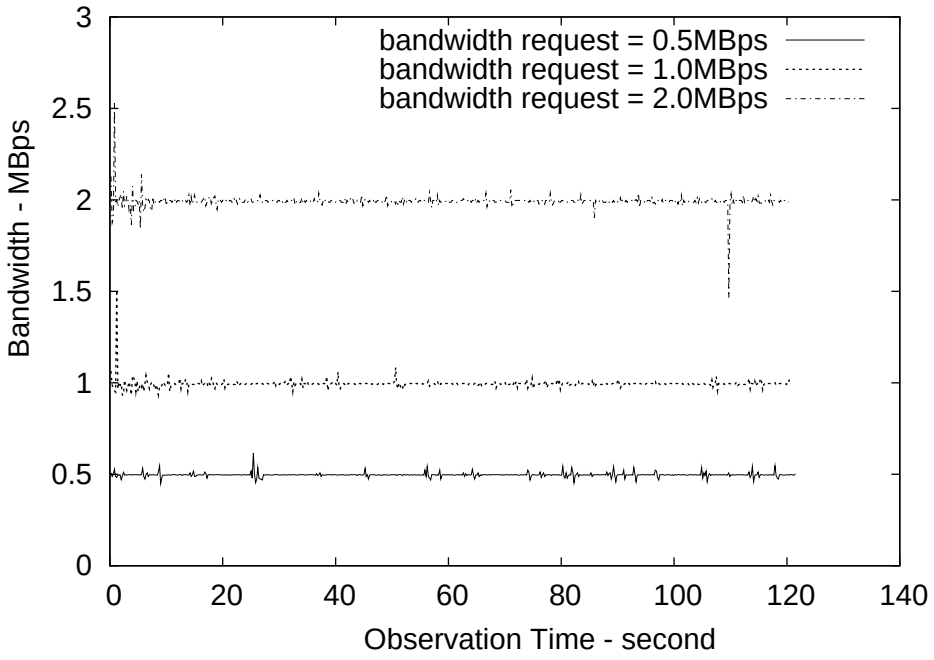

Figure 6.6: Performance of Single Flow Scheduling With 200-millisecond Time Granularity (bandwidth request: 0.5MBps-2MBps; observation time: 2 minutes; environment: two-node)

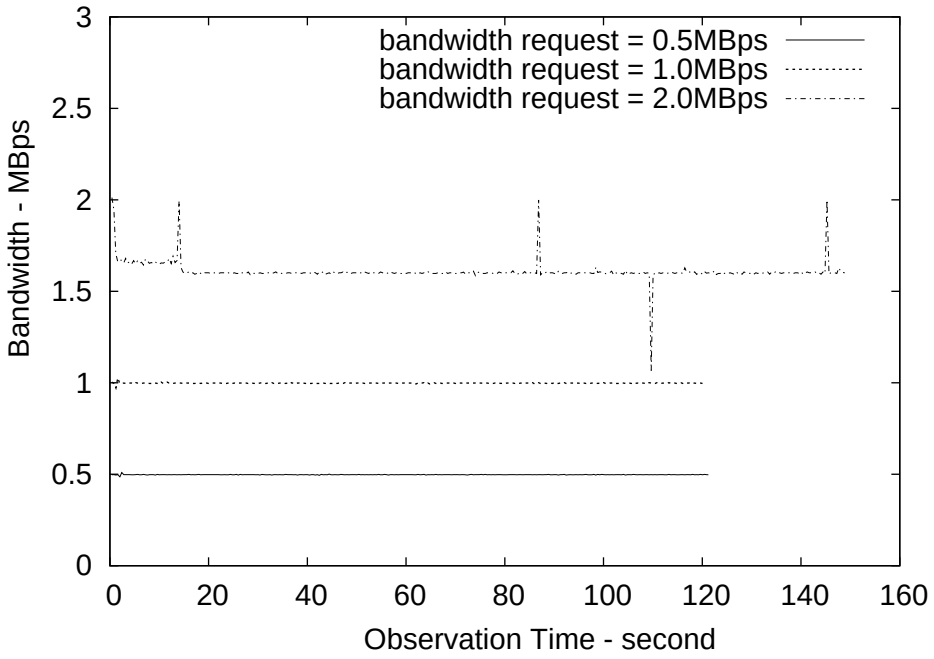

Figure 6.7: Performance of Single Flow Scheduling With 400-millisecond Time Granularity (bandwidth request: 0.5MBps-2MBps; observation time: 2 minutes; environment: two-node

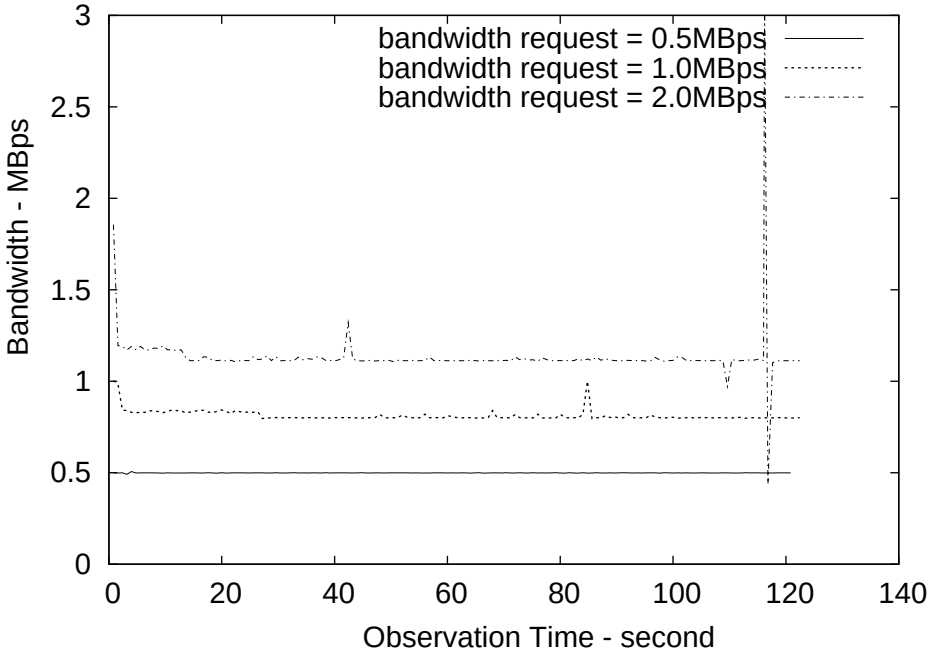

Figure 6.8: Performance of Single Flow Scheduling With 800-millisecond Time Granularity (bandwidth request: 0.5MBps-2MBps; observation time: 2 minutes; environment: two-node)

Table 6.2: Inaccuracy Comparison of Scheduling With Different Time Granularities

Time Granularity	200 ms	400 ms	800 ms
Inaccuracy(bandwidth request = 0.5MBps)	0.855	0.157	0.097
Inaccuracy(bandwidth request = 1MBps)	0.801	0.149	18.925
Inaccuracy(bandwidth request = 2MBps)	0.583	19.501	43.945

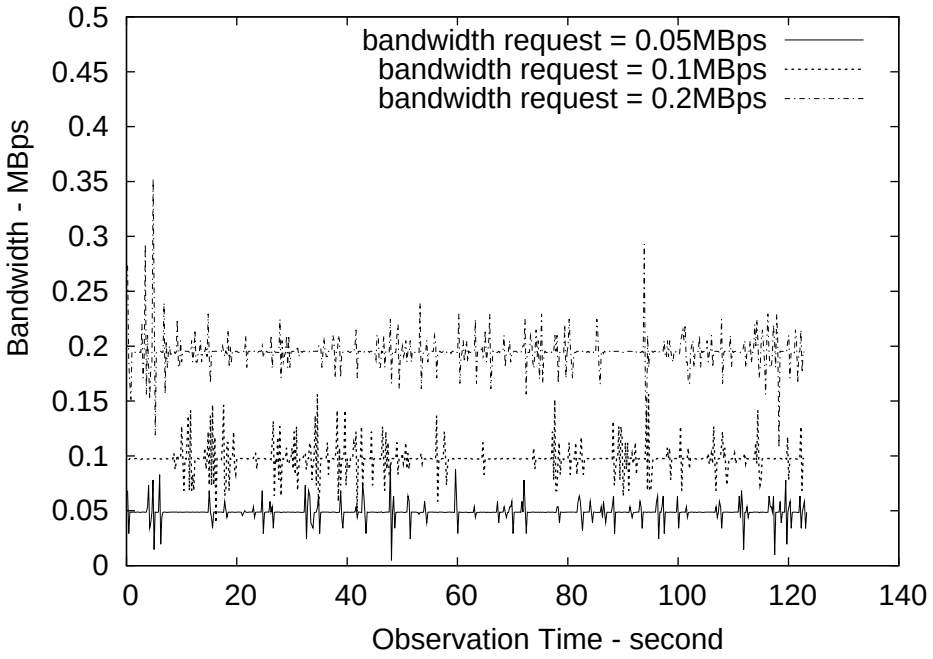

Figure 6.9: Performance of Bandwidth Allocation With 1 Flow In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: two-node)

not hold for 1MBps and 2 MBps requests. As shown in Table 6.2, the inaccuracy decreases when the time granularity increases from 200ms to 400 ms for 1MBps bandwidth request, but increases after the time granularity hits 800ms. As to 2MBps request, the inaccuracy keeps increasing. In this case, packet loss is the major factor causing inaccuracy increment. With a fixed bandwidth request, when time granularity is increased, the total number of packets to be scheduled in each cycle increases. When the number is around 500 (observed from experiment), serious packet loss happens. This result reflects that when the bandwidth request is fixed, the increment of time granularity can reduce inaccuracy in allocation only if the bulk size packets scheduled in each cycle does not cause serious packet loss (500 in this experiment).

Scheduling With Different Number of Flows In the third experiment, the number of co-existing flows from 1, 5 to 10 is increased to test the influence of this change on scheduling performance. The same experiment is repeated for different bandwidth requests. The time granularity is still 200 milliseconds.

Figure 6.9, Figure 6.10 and Figure 6.11 exhibit bandwidth allocation results for multi-flow scheduling and inaccuracy analysis is shown in Table 6.3. The result reveals that the increment of inaccuracy is not substantial when the total number of flows under scheduling increases. As in Table 6.3, the inaccuracy does not increase linearly with the number of flows. However, similar to

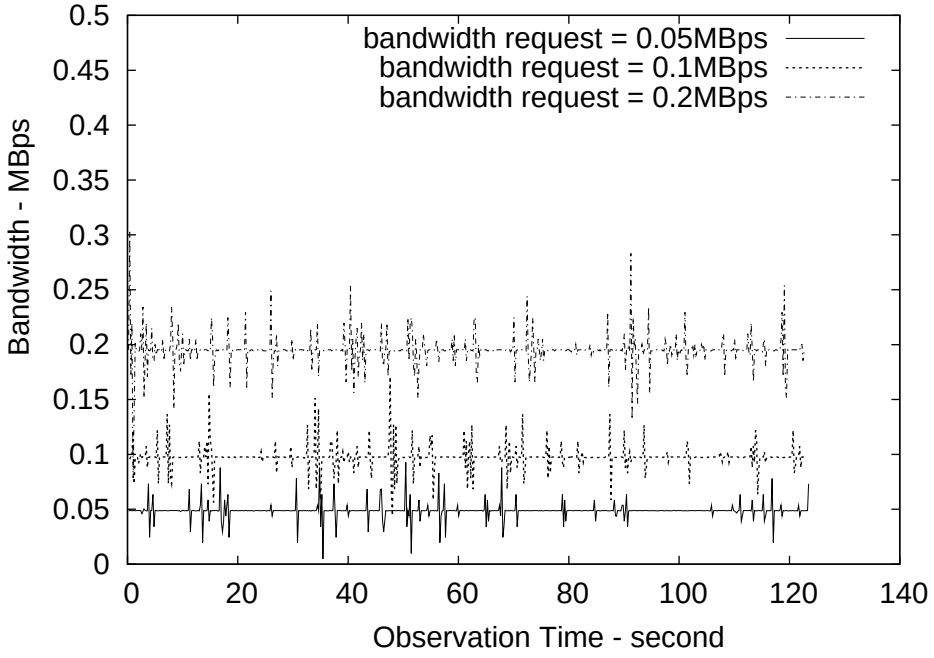

Figure 6.10: Performance of Bandwidth Allocation With 5 flows In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: two-node)

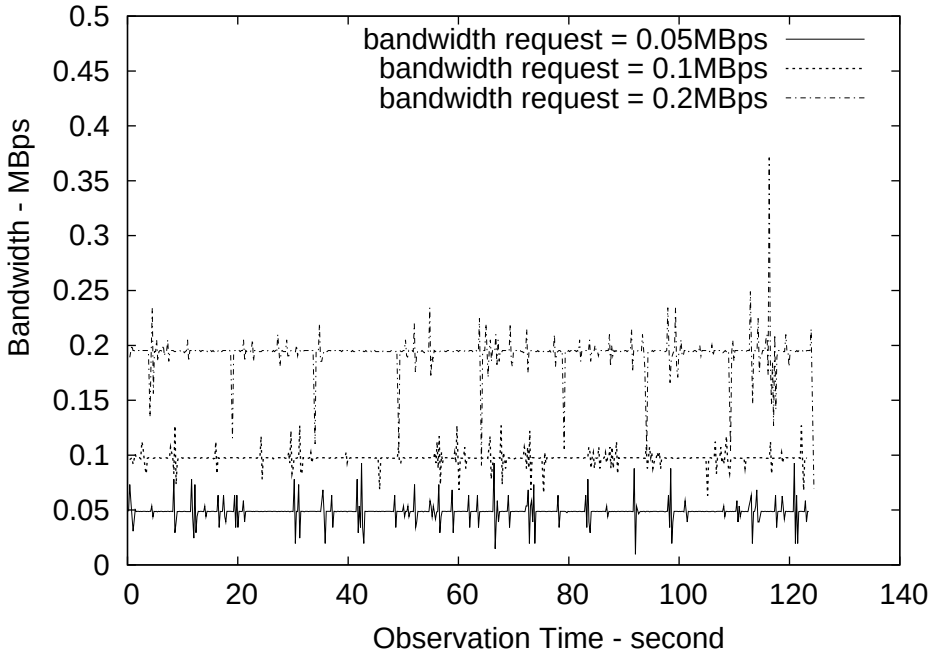

Figure 6.11: Performance of Bandwidth Allocation With 10 flows In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: two-node)

Table 6.3: Inaccuracy Comparison of Scheduling With Different Number of Flows

Number of Flows	1 flow	5 flows	10 flows
Inaccuracy(bandwidth request = 0.05MBps)	5.894 %	5.08 %	6.695 %
Inaccuracy(bandwidth request = 0.1MBps)	6.187 %	4.236 %	2.314 %
Inaccuracy(bandwidth request = 0.2MBps)	4.869 %	4.512 %	4.312 %

Table 6.4: Inaccuracy Comparison of Three-node Scheduling and Two-node Scheduling

Bandwidth request	0.5 MBps	2 MBps	5 MBps
Inaccuracy(in three-node environment)	1.057 %	0.546 %	48.301 %
Inaccuracy(in two-node environment)	0.855 %	0.58 %	31.119 %

the analysis given in the time granularity experiment, this only indicates before the total number of packets being sent out in each cycle does not cause serious packet loss, the performance of bandwidth allocation is relatively stable when the number of flows increases.

6.2.2 Three-Node Experiment

In the three-node environment, experiments of single flow scheduling with different bandwidth requests, time granularities and multi-flow scheduling are repeated. These experiments aim to test the influence of the intermediate node on scheduling performance.

Single Flow Scheduling With Different Bandwidth Requests

In the scenario of single flow scheduling with different bandwidth requests, a single flow requires bandwidth from 0.5MBps to 5 MBps and the time granularity of each scheduling cycle is 200 milliseconds.

Figure 6.12 shows the similar pattern as that in the two-node experiment. When the bandwidth request is less than 2.5 MBps, the allocated bandwidth is close to the requested value, but after the request exceeds 2.5 MBps, the allocated bandwidth is significantly deviated from the requested value. Inaccuracy is presented in Table 6.4. Besides, the three-node environment causes more packet loss, because the intermediate node reschedules received packets. Without rescheduling, in-transit packets are sent out upon being received. This may interfere the scheduling of packets originated from the intermediate node and results in packet scheduling out of control.

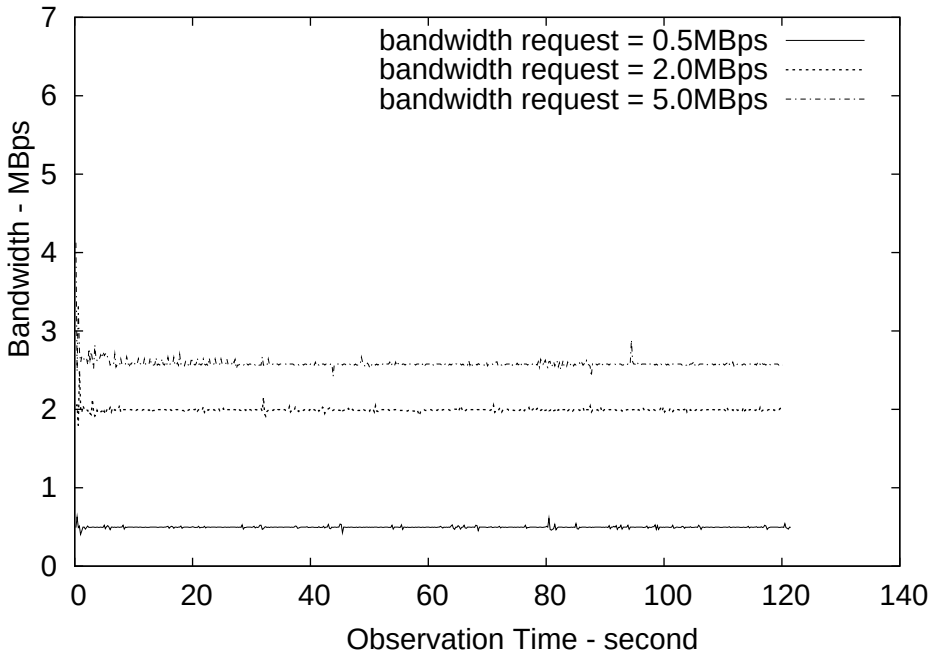

Figure 6.12: Performance of Bandwidth Allocation With Different Bandwidth Requests (number of flows: 1; bandwidth request: 0.5MBps-5MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: three-node)

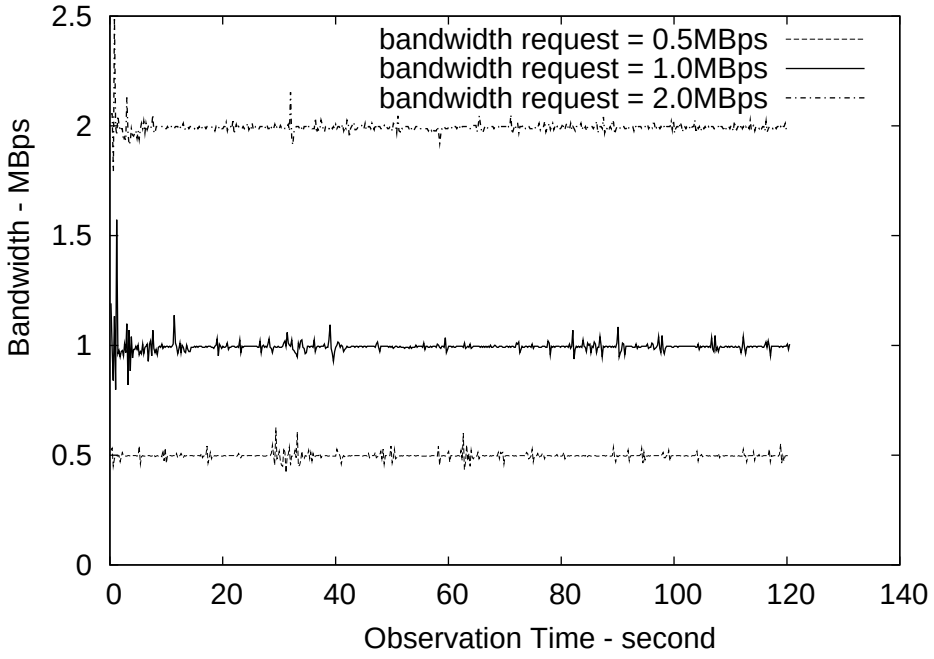

Figure 6.13: Performance of Single Flow Scheduling With 200-millisecond Time Granularity (bandwidth request: 0.5MBps-2MBps; observation time: 2 minutes; environment: three-node)

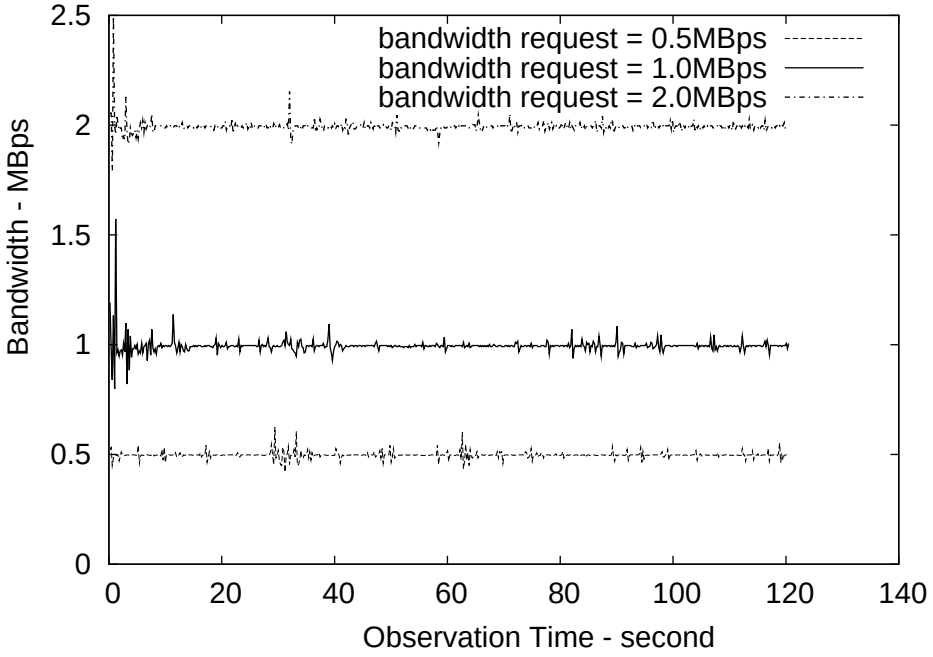

Figure 6.14: Performance of Single Flow Scheduling With 800-millisecond Time Granularity (bandwidth request: 0.5MBps-2MBps; observation time: 2 minutes; environment: three-node)

Table 6.5: Inaccuracy Comparison of Scheduling With Different Time Granularities in Three-Node Environment

Time Granularity	200 ms	400 ms	800 ms
Inaccuracy(bandwidth request = 0.5 MBps)	1.35 %	0.152 %	0.22 %
Inaccuracy(bandwidth request = 1.0 MBps)	1.152 %	0.156 %	38.687 %
Inaccuracy(bandwidth request = 2.0 MBps)	0.546 %	39.599 %	63.825 %

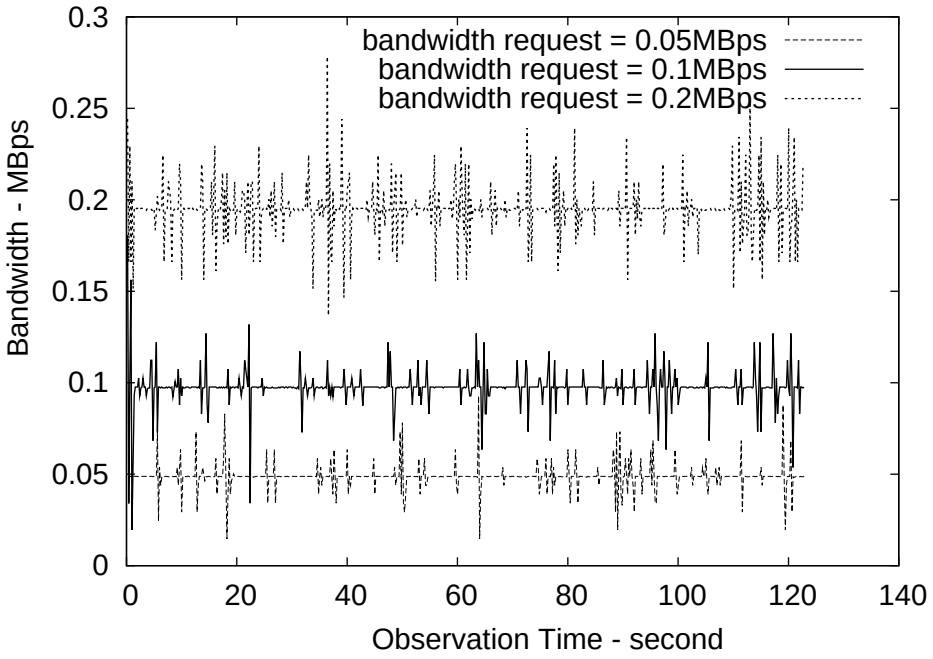

Figure 6.15: Performance of Bandwidth Allocation With 1 Flow In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: three-node)

Table 6.6: Inaccuracy Comparison of Scheduling With Different Number of Flows In Three-Node Environment

Number of Flows	1 flow	5 flows	10 flows
Inaccuracy(bandwidth request = 0.1MBps)	3.739 %	4.011 %	5.364 %
Inaccuracy(bandwidth request = 0.2MBps)	3.51 %	4.231 %	3.294 %

Scheduling With Different Time Granularities Figure 6.13 and Figure 6.14 compares bandwidth allocation for a single flow with 200 and 800 milliseconds time granularity. For the request of 0.5 MBps, the larger the time granularity is, the more stable the bandwidth allocation achieves. By contrast, for bandwidth allocation requests of 1 MBps and 2 MBps, the larger the time granularity, the worse the bandwidth allocation performance. This shows the similar pattern as that in the two-node experiment. Inaccuracy analysis is described in Table 6.5

Scheduling With Different Number of Flows In the multi-flow scheduling, the 1 flow case is compared with the 10 flow scenario. As shown in Table 6.6, the difference between bandwidth allocation in these two cases is not significant and this demonstrates that the bandwidth allocation of an individual flow is guaranteed without influences from coexisting flows.

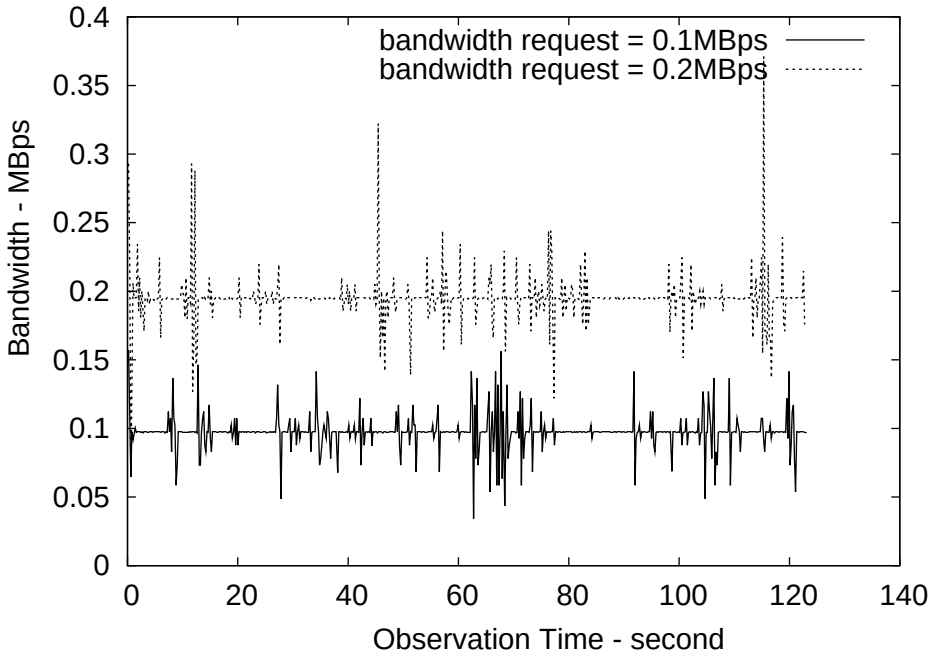

Figure 6.16: Performance of Bandwidth Allocation With 10 Flows In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: three-node)

6.3 Multi-Node Simulation

In order to further validate the pattern of scheduling performance gained from two-node and three-node experiments, a multi-node simulation is used to test bandwidth allocation performance in multiple-node environment. As shown in Figure 6.3, the simulated environment has 7 nodes (1, 2, 3, 4, 5, 6 and 7). This is composed by creating 3 loops between node S and node M, and node D serves as the destination node. According to this configuration, node S simulates node 1, 3 and 5, and node M simulates node 2, 4 and 6. In each cycle, node S has to schedule packets for simulated node 1, 3 and 5, and so does node M. This means the total number of packets to be sent in each round in the simulated environment is 3 times than the actual 7 node environment. As a result, the simulated environment introduces extra influence on scheduling and significant packet loss happens with lower bandwidth requests, smaller time granularities and less number of flows. With these concerns, lower requests for bandwidth, time granularity are used in this experiment.

Single Flow Scheduling With Different Bandwidth Requests

In this scenario, a single flow requests bandwidth ranging from 0.5MBps to 1.5 MBps and the time cycle is 200 milliseconds.

Results shown in Figure 6.17 illustrates that when the bandwidth request is less than or equal to

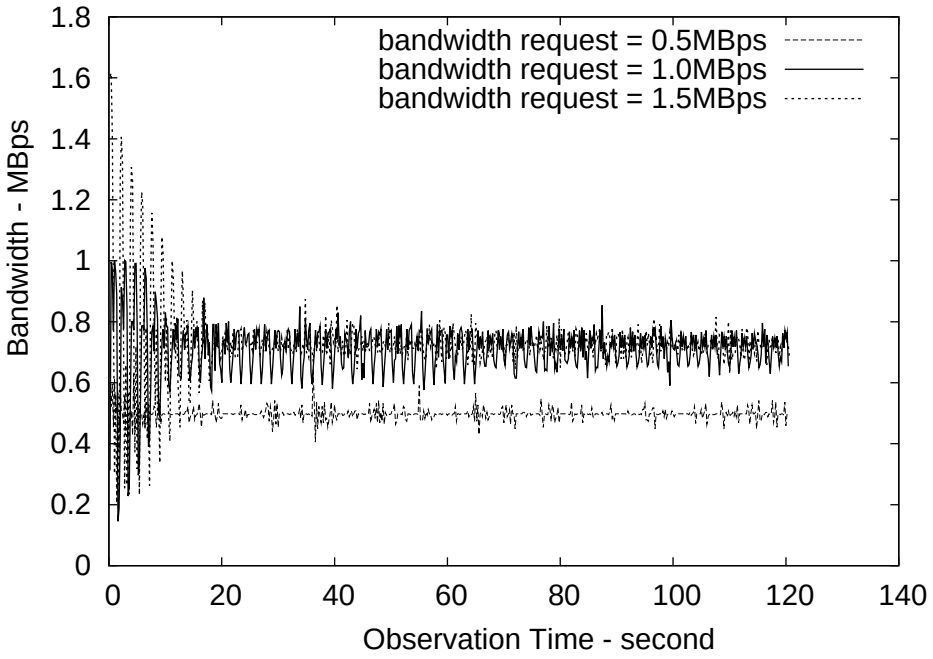

Figure 6.17: Performance of Bandwidth Allocation With Different Bandwidth Requests (number of flows: 1; bandwidth request: 0.5MBps-1.5MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: multi-node)

0.8 MBps (in the simulation, node 1 and 2 has to schedule 491 packets respectively in each cycle), the bandwidth allocation is still stable. Beyond this value, the bandwidth allocation significantly deviates from the requested value.

Scheduling With Different Time Granularities In this experiment, time granularities of 100 ms, 200 ms and 300 ms are used to test bandwidth allocation for bandwidth request of 0.5 MBps, 1.0 MBps and 1.5 MBps.

Performances of scheduling with 100 ms and 300 ms time granularities are depicted in Figure 6.18 and Figure 6.19. Table 6.7 shows that for the bandwidth request of 0.5 MBps, the increment

Table 6.7: Inaccuracy Comparison of Scheduling With Different Time Granularities in Multi-Node Simulation

Time Granularity	100 ms	200 ms	300 ms
Inaccuracy(bandwidth request = 0.5 MBps)	1.03 %	2.282 %	8.937 %
Inaccuracy(bandwidth request = 1.0 MBps)	0.895 %	28.557 %	52.015 %
Inaccuracy(bandwidth request = 1.5 MBps)	7.365 %	51.47 %	66.395 %

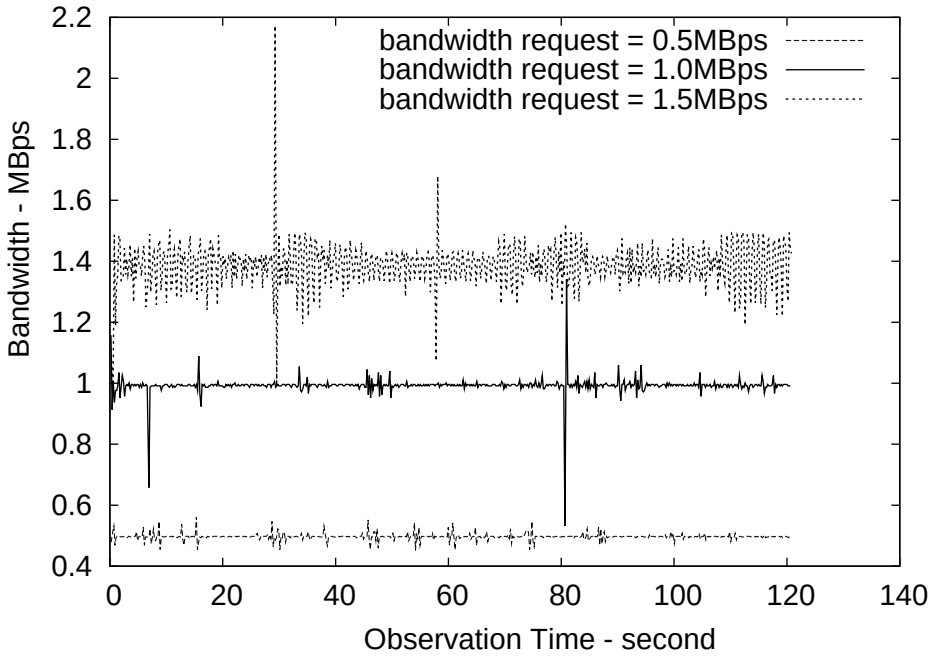

Figure 6.18: Performance of Single Flow Scheduling With 100-millisecond Time Granularity (bandwidth request: 0.5MBps-1.5MBps; observation time: 2 minutes; environment: multi-node)

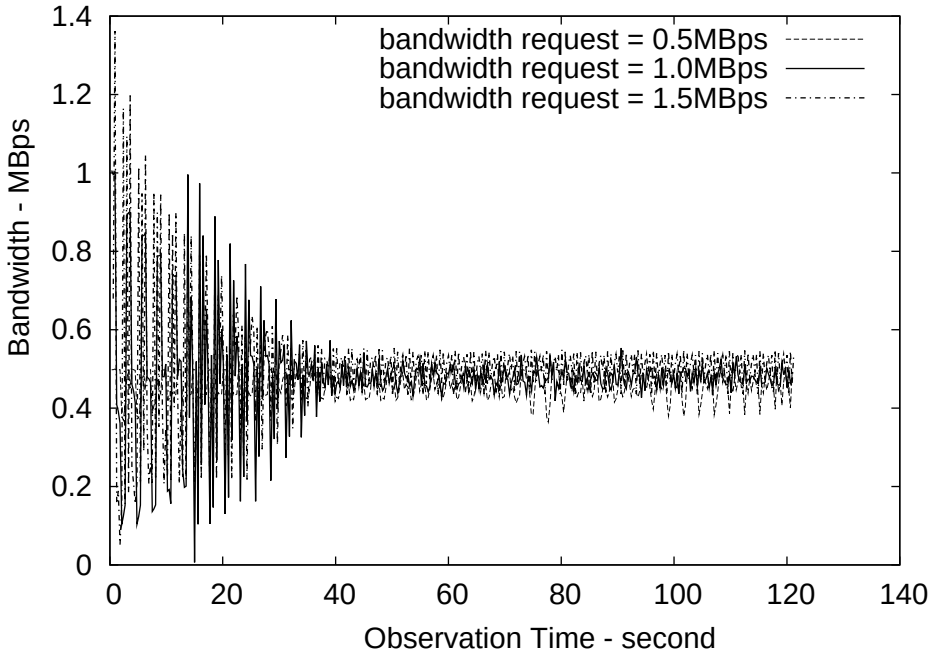

Figure 6.19: Performance of Single Flow Scheduling With 300-millisecond Time Granularity (bandwidth request: 0.5MBps-1.5MBps; observation time: 2 minutes; environment: multi-node)

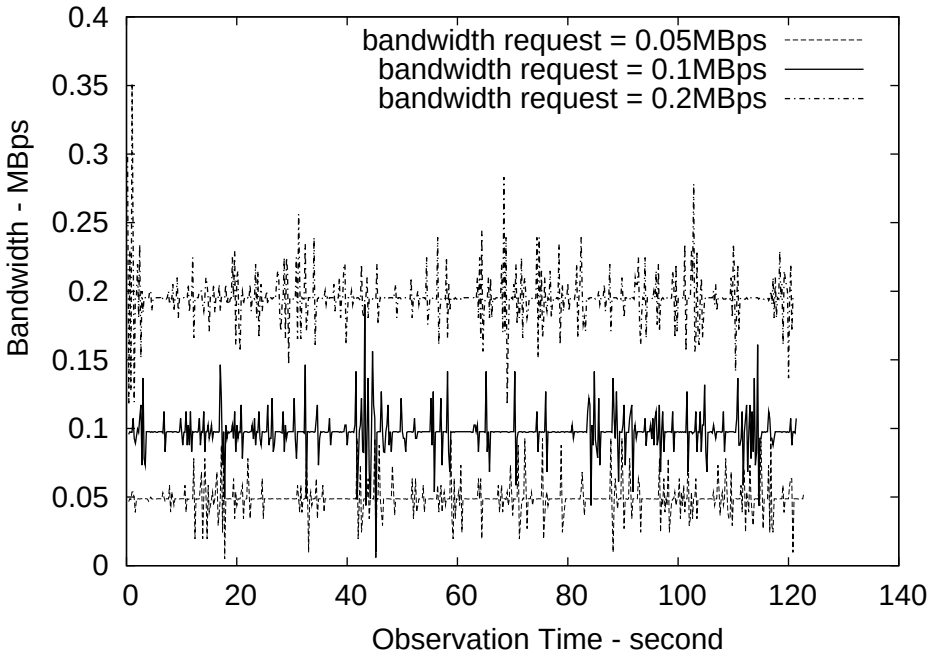

Figure 6.20: Performance of Bandwidth Allocation With 1 Flow In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: multi-node)

of time granularity does not significantly increase the inaccuracy in bandwidth allocation. Because of the simulation limitation described at the beginning of this section, this result does not show the trend observed in two-node and three node experiments, which is larger the time granularity is, the more stable bandwidth is allocated. Even in this case, the increment of time granularity does not introduce significant inaccuracy (more than 10%).

Scheduling With Different Number of Flows In this scenario, experiments are performed to compare bandwidth allocation performance between scheduling with 1 flow and 10 flows.

As shown in Figure 6.20 and Figure 6.21, the difference between these two cases is not significant. However, this is only true when significant packet loss does not happen. By observation, after the significant packet loss happen, flows which are scheduled at the end of each cycle are influenced seriously. Figure 6.22 shows the worst scheduling performance for the 10 flow case.

6.4 Scheduling Overhead Analysis

Scheduling overhead of this scheme is resulted from multiple queue switching, thread context switching, message partitioning, packet sending and receiving, and is measured by accounting the average processing time dedicated to packet and message scheduling in each cycle and the total number of

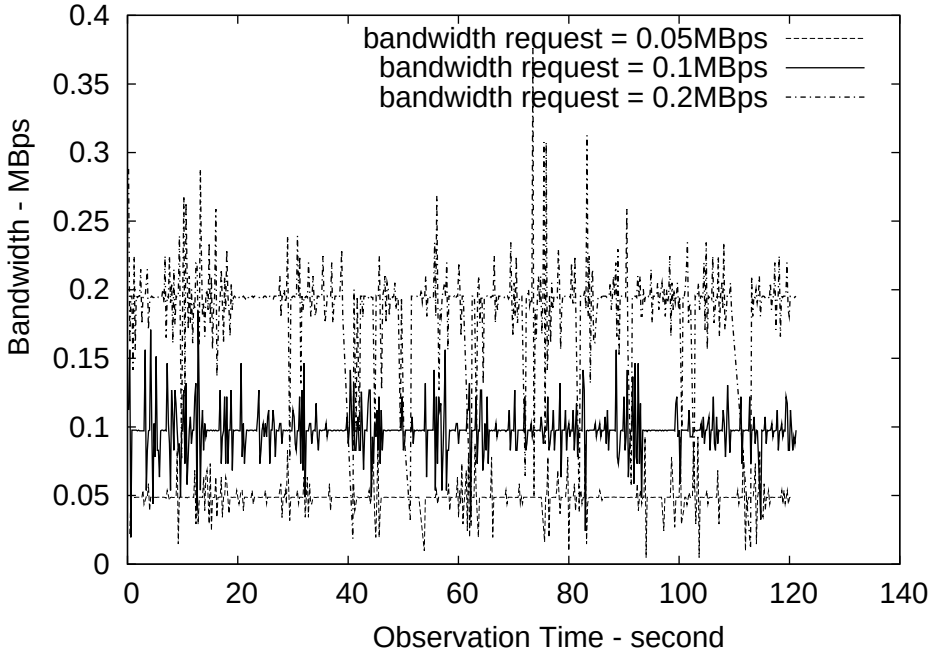

Figure 6.21: Performance of Bandwidth Allocation With 10 Flows In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: multi-node)

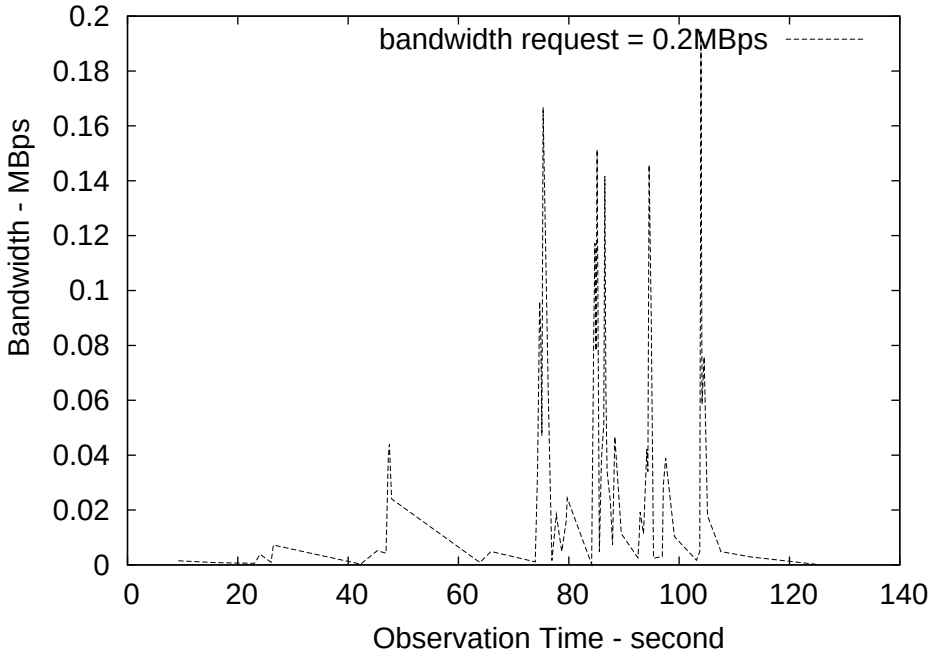

Figure 6.22: Worse Performance of Bandwidth Allocation With 10 Flows In System (bandwidth request: 0.05MBps-0.2MBps; time granularity: 200 millisecond; observation time: 2 minutes; environment: multi-node)

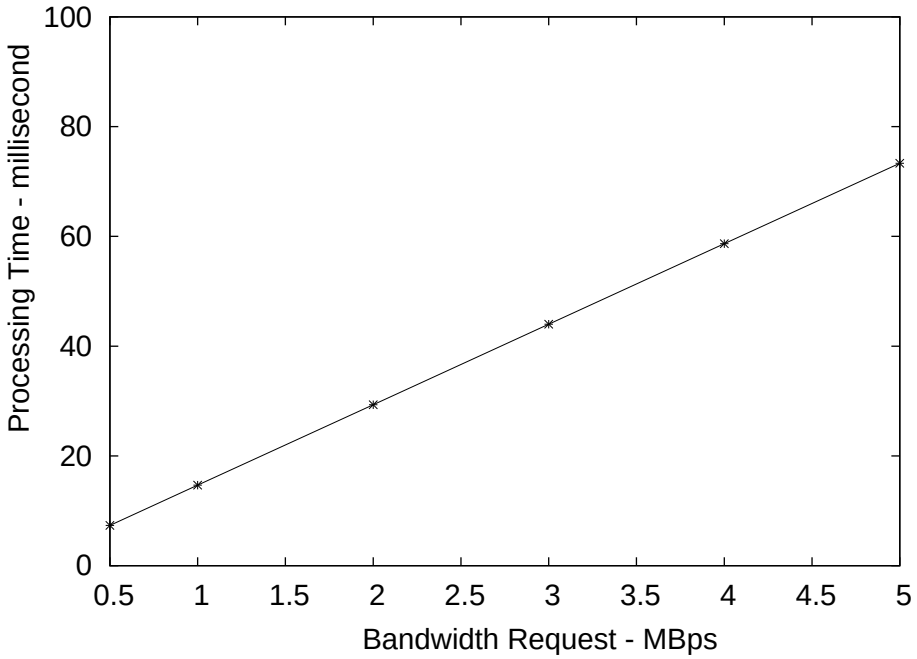

Figure 6.23: Scheduling Overhead (number of flows: 1; bandwidth request: 0.5MBps-5MBps; time granularity: 200 millisecond; environment: two node)

cycles is 400.

Table 6.8 and Table 6.9 shows overhead comparison of scheduling for 0.1MBps request and 0.2MBps request respectively. In this case, 1, 5 and 10 flows are scheduled with simple scheduling, CyberOrgs scheduling[2] in two-node and CyberOrgs scheduling in multi-node environment. The simple scheduling scheme schedules different flows in a single queue and controls the total bandwidth allocation to all these flows. Corresponding overhead of simple scheduling includes the processing time for message partitioning, and single packet scheduling. The two node processing time with CyberOrgs scheduling adds extra cost for multi-queue switching. In the multi-node scheduling,

[2]CyberOrgs scheduling denotes the scheduling scheme developed for the prototype system

Table 6.8: Overhead Comparison of Scheduling With CyberOrgs Scheduling Scheme and Simple Scheduling Scheme (bandwidth request: 0.1MBps; time granularity: 200 milliseconds)

Number of Flows	1 flow	5 flows	10 flows
Processing Time (With Simple Scheduling)	1.488 ms	5.468 ms	8.95 ms
Processing Time (CyberOrgs Scheduling in Two-Node)	2.868 ms	7.905 ms	14.168 ms
Processing Time (CyberOrgs Scheduling in Multi-Node)	4.31 ms	16.81 ms	27.6 ms

Table 6.9: Overhead Comparison of Scheduling With CyberOrgs Scheduling Scheme and Simple Scheduling Scheme (bandwidth request: 0.2MBps; time granularity: 200 milliseconds)

number of flows	1 flow	5 flows	10 flows
Processing Time (With Simple Scheduling)	2.2 ms	9 ms	16.8 ms
Processing Time (CyberOrgs Scheduling in Two-Node)	3.8 ms	13.5 ms	28.2 ms
Processing Time (CyberOrgs Scheduling in Multi-Node)	5.045 ms	17.05 ms	32.14 ms

each node has roles of source, intermediate and destination, processing time includes message partitioning, multi-queue packet scheduling and packet receiving as well.

In addition, the experiment data collected shows that some scheduling cycles run significantly longer time than the specified time granularity and this phenomenon happens periodically (almost every 75 cycles). This can be caused by Java language which is not designed for real-time purpose. As described in [45] data structure like HashMap may rehash during the computation and causes burst of execution time.

6.5 Experiment Conclusion

From the above experiments, it is concluded that the implemented scheduling scheme is effective to achieve fine-grained per-flow rate-based bandwidth allocation when the packet bulk size determined by the time granularity and bandwidth request does not cause serious packet loss.(By observation around 500 packets).

Within this validity scope, this scheduling scheme achieves adjustable time granularity, effective multi-flow scheduling and stable bandwidth allocation. Specifically, bandwidth allocation is stable for different bandwidth requests under a fixed time granularity; the larger the time granularity is, the more stable the bandwidth allocation is; bandwidth allocation to individual flow is guaranteed without significant influence from coexisting flows; and the intermediate node does not introduce obvious influence on performance.

Beyond this validity scope, the increment of bandwidth request and time granularity causes more packet loss and allocated bandwidth is significantly deviated from requested value. In addition, when there are more flows coexisting in the system, a few flows scheduled at the end of each cycle are punished with more packet loss. Furthermore, the intermediate node rescheduling also worsens scheduling performance by causing more packet loss.

From the above conclusion, this scheduling scheme can be improve several ways. The first approach is to adjust the request of bandwidth request and time granularity. With higher request,

the scheduling cycle has to be adjusted to finer granularity in order to reduce the number of packets in each cycle and achieve better performance. The second approach is to add flow control in this scheduling scheme to control packet loss. Compared with TCP, this scheduling scheme is sender-paced, which means the sender side controls the way of sending packets. TCP, by contrast is receiver-paced. Because the recipient knows when its buffer gets full, it is the recipient that informs the sender side to slow down packet sending rate. In addition, the scheduling scheme can also be improved by controlling how packets are sent during each cycle. Instead of sending all packets as a bulk, appropriate interval can be inserted between packet sending. Finally, the system can be further improved by using real-time programming in order to guarantee that each scheduling cycle finish its task on time.

Chapter 7

Conclusion and Future Work

Development of the Internet makes it possible for software systems to operate in an open and distributed environment. Multi-agent systems, consisting of distributed and asynchronous agents, which are open to interact with the environment offers a new paradigm to create systems that operate in open and distributed environments. Coordination between a set of agents, which are running in a resource sharing environment addresses the importance of having resource management mechanisms in multi-agent systems.

This research is motivated by network resource contention problems existing in multi-agent systems. As in a distributed environment, agents are spread out over network and need to communicate with each other. Network resource is consumed for these communications. However, network resource is bounded in time and space, unrestricted contention for finite network resource may have negative influence on agent communication and cause network overload and traffic congestion. This work offers a new approach to reify network resource control in multi-agent systems and implements an efficient prototype system.

In this work, the general resource management model CyberOrgs is specialized in network resource control. This specialization introduces a new concept vLink to abstract network resource. vLink characterizes time, spatial and QoS related attributes of network resource, rather than concrete configuration. This abstraction offers a simplified and uniform representation of network resource, by which programmers are enabled to request network resource in terms of a set of attributes. Corresponding mechanisms of vLink generation, allocation and consumption are developed to coordinate network resource usage among agents. In addition, by separating resource concerns from functional concerns of computation, programmers are enabled to express specific resource requests for their applications and resource control policy separately from application development. The prototype system implements the CyberOrgs model in a distributed manner and provides APIs for programmers to control network resource. An efficient scheduling scheme is also developed to enable fine-grained, flow-based bandwidth allocation. The hierarchical resource schedule in CyberOrgs is implemented as a flat structure to reduce prohibitive cost of hierarchical scheduling.

7.1 Contribution

This work achieves contributions in the area of network resource management, as follows:

- Specializing CyberOrgs in network resource control
- Developing and implementing network resource abstraction Virtual Link
- Enabling an effective fine-grained per-flow rate-based bandwidth allocation scheme
- Providing a prototype system to reify hierarchical network resource control in multi-agent systems

7.2 Future Work

This work is the first attempt of specializing CyberOrgs in network resource control and leaves space for further improvement. First of all, network resource is controlled in a closed system at the current stage in order to predict available resource. This assumption is too strict for real system application, and needs to be loosened. Secondly, in the current system, network resource discovery is local to a cyberorg. In the future work a scalable and efficient mechanism for global resource discovery is worth of exploring. Next, due to time limit current experiments are carried out in a simple network configuration. In the future, a larger system can be built to operate the prototype system and explore performance issues caused by more complex scenarios.

References

[1] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artifical Intelligence*. Addison-Wesley, 1999.

[2] N. Jamali. *CYBERORGS: A Model for Resource Bounded Complex Agents*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[3] C. Hewitt and P. Jong. Open Systems. In J. W. Schmidt J. Mylopoulos and M. L. Brodie, editors, *On Conceptual Modelling*, pages 147–164. Springer Verlag, 1984.

[4] L. Gasser. DAI Approaches to Coordination. In N. M. Avouris and L. Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 31–51. Kluwer-Academic, 1992.

[5] M. N. Huhns and L. M. Stephens. Multiagent Systems and Societies of Agents. In *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 79–120. MIT press, 1999.

[6] G. Czajkowski and T. V. Eichen. Jres: A Resource Accounting Interface for Java. In *Proceedings of the 1998 ACM OOPSLA Conference*, pages 21–35, Octobor 1998.

[7] W. Binder, J. G. Hulaas, and A. Villazon. Portable Resource Control in Java: The J-seal2 Approach. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 139–155, Octobor 2001.

[8] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A Resource Management API for Java Platform. Technical Report SMLI TR-2003-124, 2003.

[9] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, and T. S. Mitrovich. An overview of the nomads mobile agent system. In *Proceedings of 2000 ECOOP Workshop On Mobile Object Systems*, June 2000.

[10] L. Moreau and C. Queinnec. Design and semantics of quantum: a language to control resource consumption in distributed computing. In *Proceedings of the 1997 Usenix Conference on Domain-Specific Languages*, pages 183–197, Octobor 1997.

[11] L. Moreau and C. Queinnec. Distribued and multi-type resource management. In *Proceedings of the 2002 ECOOP Workshop on Resource Management for Safe Languages*, pages 1–14, June 2002.

[12] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. 15(3):200–222, 2001.

[13] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor:a hunter of idle workstations. In *In Proceedings of the 1988 IEEE ICDCS Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[14] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[15] I. Foster and C. Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(6):607–621, 1999.

[16] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan-Kaufmann, 2000.

[17] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the 1988 ACM SIGCOMM Conference*, pages 314–329, August 1988.

[18] M. Allman and V. Paxson. TCP Congestion Control. RFC 2581, April 1999.

[19] C. David, S. Scott, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms. In *Proceedings of the 1992 ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 14–26, August 1992.

[20] V. Jacobson. Differentiated Services for the Internet. In *Proceedings of First Internet2 Joint Applications/Engineering QoS Workshop*, pages 26–31, May 1998.

[21] S. Ghosh. *Scalable QoS-based Resource Allocation*. PhD thesis, Carnegie Mellon University, 2004.

[22] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, June 1994.

[23] C. Partridge. A Proposed Flow Specification. RFC 1363, Septmeber 1994.

[24] C.Topolcic. Experimental Internet Stream Protocol: Version 2 (ST-II). RFC 1190, October 1990.

[25] S. Jamin, P. B. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. In *Proceedings of the 1995 ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication*, pages 56–70, January 1995.

[26] L. Zhang, S. Deering, and D. Estrin. RSVP: A New Resource ReSerVation Protocol. *IEEE network*, 7(5):8–30, September 1993.

[27] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification. RFC 2205, September 1997.

[28] S. Blake, D. L. Black, M. A. Carlson, and R. Morris. An Architecture for Differentiated Services. RFC 2475, October 1998.

[29] B. Davie, A. Charny, J.C.R. Bennet, K. Benson, J.Y. Le Boudee, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An Expedited Forwarding PHB (Per-Hop Behavior). RFC 3246, March 2002.

[30] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. RFC 2597, June 1999.

[31] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[32] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay based Architecture for Enhancing Internet QoS, March 2004.

[33] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D.J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[34] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the 1998 International Conference on Functional Programming*, pages 86–93, September 1998.

[35] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[36] A. K. Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Massachusetts Institute of Technology, February 1992.

[37] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[38] C. Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the 1969 International Join Conference in Artificial Intelligence*, pages 295–301, May 1969.

[39] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence IJCAI. In *Proceedings of the 1973 International Joint Conference on Artificial Intelligence*, pages 235–245, August 1973.

[40] Open System Laboratory. The Actor Architecture. Technical report, University of Illinos at Urbana-Champaign, 2004.

[41] N. Jamali and X. Zhao. A scalable approach to multi-agent resource acquisition and control. In *Proceedings of the 2005 international joint conference on Autonomous agents and multiagent systems*, pages 868–875, July 2005.

[42] T. W. Malone. Modeling coordination in organizations and markets. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 1317–1332, 1988.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *An Introduction to Algorithms, Second Edition*. MIT Press, 2001.

[44] NM. Jain and C. Dovrolis. Pathload: A Measurement Tool for End-to-End Available Bandwidth. Proc. 3rd Passive and Active Measurements Workshop. In *Proceedings of the 2002 Passive and Active Measurements Workshop*, pages 14–25, March 2002.

[45] J. M. Dautelle. *Collection Classes for Real-Time and High-Performance Applications*. http://javolution.org/doc/Javolution-Collections.pdf/, 2005.